

LEVEL

UCLA-ENG-8114
SRPS-81-002
JULY 1981

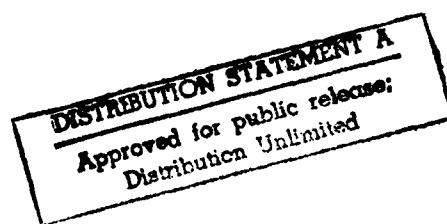
AD A103371

SECURE RELIABLE PROCESSING SYSTEMS:
SEMI-ANNUAL TECHNICAL REPORT,
JULY 1979 – JUNE 1981

102
DTIC
AUG 26 1981
H

GERALD J. POPEK, PRINCIPAL INVESTIGATOR

SECURE SYSTEMS AND SOFTWARE
ARCHITECTURE GROUP



WE FILE COPY

COMPUTER SCIENCE DEPARTMENT

School of Engineering and Applied Science
University of California
Los Angeles



81 8 26 053

6 SECURE RELIABLE PROCESSING SYSTEMS
9 SEMI-ANNUAL TECHNICAL REPORT
1 Jul 1979 — 30 Jun 1981

14 UCLA-ENG-8114,
SRPS - 81-002

12
11 Jul 81
12 215
S DTIC ELECTE D
AUG 26 1981
H

10 Gerald J. Popek
Principal Investigator
Computer Science Department
School of Engineering and Applied Science
University of California at Los Angeles
(213) 825-6507

This research was sponsored by the
Defense Advanced Research Projects Agency.

15
ARPA Contract Period: 1 July 1977 - 30 September 1981
ARPA Contract Number: DSS MDA-903-77-C-0211
ARPA Order Number: 3396
Program Code Number: 7P10
ARPA Order - 3396

The views and conclusions contained in this document are
those of the author and should not be interpreted as
necessarily representing the official policies, either
express or implied, of the Defense Advanced Research
Projects Agency or the United States Government.

DISTRIBUTION STATEMENT A
Approved for public release
Distribution Unlimited

405749

pt

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) SECURE RELIABLE PROCESSING SYSTEMS: SEMI-ANNUAL TECHNICAL REPORT, JULY 1979 - JUNE 1981		5. TYPE OF REPORT & PERIOD COVERED Semi-annual tech. report 7/1/79 - 6/30/81
7. AUTHOR(s) Gerald J. Popek, Principal Investigator		6. PERFORMING ORG. REPORT NUMBER DSS MDA-903-77-C-0211 <i>new</i>
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Science Dept.; School of Engr. & Applied Science; Univ. of Calif. at Los Angeles; Los Angeles, Calif. 90024		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS ARPA Order No. 3396 <i>new</i>
11. CONTROLLING OFFICE NAME AND ADDRESS		12. REPORT DATE July 1981
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Office of Naval Research 1030 East Green Street Pasadena, California 91101		13. NUMBER OF PAGES 215
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release. Distribution unlimited.		18. SECURITY CLASS. (of this report) Unclassified
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		18a. DECLASSIFICATION/DOWNGRADING SCHEDULE
18. SUPPLEMENTARY NOTES		ACCESSION FOR NTIS GRA&I <input checked="" type="checkbox"/> DTIC TAB <input type="checkbox"/> Unannounced <input type="checkbox"/> Justification
19. KEY WORDS (Continue on reverse side if necessary and identify by block number)		By Distribution/ Availability Codes Dist Special <i>A</i>
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) This semi-annual technical report covers research carried out by the Secure Reliable Processing Systems Contract at UCLA under ARPA Contract DSS MDA-903-77-C-0211 covering the period 1 July 1979 through 30 June 1981. This report also contains the Ph.D. Dissertation by Charles Steven Kline (Co-Principal Investigator) "Data Security: Operating Systems and Computer Networks", conducted under the supervision of Professor Gerald J. Popek (Principal Investigator for this research).		

SECURE RELIABLE PROCESSING SYSTEMS

Advanced Research Projects Agency Semi-Annual Technical Report

July 1979 - June 1981

1. RESEARCH ACTIVITIES

The current contract activities have been organized into three tasks:

1. Operating Systems and Applications Security
2. Network Security
3. Reliable Distributed Systems Architectures

with most emphasis on task three. Below, we review the progress that was made during the current contract period in each of these areas.

1.1. Operating Systems and Applications Security

During the past several years, a large portion of our research involved development of methodology for the construction and verification of secure systems. As part of that effort, a prototype secure operating system, the UCLA Data Secure Unix System, was constructed and largely verified [POPE79a][KEMM79].

In the current contract period, two M.S. theses dealing with that work were published, completing and substantially reporting that work [KAMPB1][URBA79]. The first described the design and implementation of the Unix interface package which sits on top of the secure kernel. That package must transform the somewhat awkward security kernel interface to a more reasonable one, and do so in an efficient manner. The second thesis described the design of the file/policy manager module. It is that module which implements security policy, including file access protection.

In the UCLA Secure Unix system, a new model of access protection was used based on information flow concepts. In the model, users are able to control and keep track of how information in their files moves around the system. The notion is that users can tag files with labels called colors. If some user wishes to write a file after having read some files, the colors of the written file must be a superset of the colors of the read files. When they are not, either the write attempt is blocked or the written file

colors are updated to be the union of the read file colors. The exact choice is left up to the owner of the written file. Thus, the specification of the conditions under which the access attempt should block and the conditions where the colors should propagate is part of the user visible protection model.

That work has been extended recently and a new model, access flow, has been developed. The access flow model attempts to better explain and understand the relationships of access control and information flow. In particular, the access flow model not only controls access to objects and information flow among objects, but also controls the flow of access rights among objects. As an example, with the new access flow tools one can pass user A the right to read an object but with user B denied that right. Nothing user A can do can pass that information without also passing the denial of the right of user B to access the information. An M.S. thesis is currently being completed in this area [STOU81].

In the previous contract period, the UCLA Security Kernel was verified in a doctoral dissertation by Kemmerer [KEMM79]. That work utilized the formal abstract type and mapping techniques of Alphard, and demonstrated that they were suitable as a basis for formal verification of security properties. A high level model of security was presented along with high level specifications and invariants for the system. A function was then presented enabling mapping to the more concrete representations used in the code. Verifications were performed at the concrete level. As usual with such methods, there is a great deal of detail since the approach was oriented to machine proofs; hence formal descriptions and proof steps are exceedingly complete. We expect efforts such as this one to continue to move verification methodology to the point where these large proof tasks can be supported by some form of automated aids.

However, there is still a need for more intuitive approaches, both because the machine technology is not yet present, but also because verification shows consistency of a model of the system with its formally stated specifications; errors are possible both in preparing the model and specifications and in the proofs, and the detail makes human review difficult.

Recognizing these needs, during the current contract a different, intuitive based proof methodology was developed. Kline presents in his doctoral dissertation a more user understandable model of security and shows how the proofs of the security of the UCLA system could be performed in this more intuitive, user understandable way [KLIN80].

The two approaches complement each other quite well.

The security proofs are, of course, quite similar in principle, once the formal models are understood. Specifications and invariants in one model naturally can be found (possibly in a different form) in the other. These two companion methods form, in our opinion, a powerful contribution to security certification methodology.

1.1.1. Database System Security

While operating system security still presents significant research issues, we believe that substantial progress has been achieved. Prototype secure systems have been constructed; work has been progressing on commercial versions.

However, many harder and more important problems remained, especially in database system security. These systems are larger and more complex than operating systems, and even less was known about implementing effective security measures.

After examining several database management systems, it appeared that security kernel technology might be successfully applied to certain database management problems: in particular, data independent security control, i.e. when access control decisions do not depend on stored application data values. This particular case is of considerable practical import, since most of the existing data management security needs seem to be expressible in this form.

Once again, the goal was a small, verifiable kernel which mediates all access attempts. In that way, the massive amount of complex software composing the major portion of the database system would not need to be trusted. Therefore, an effort was started to study, design, and implement a prototype database security kernel for the Ingres database system.

The data independent case results in considerable simplification because none of the data management software involved in actually computing, locating, and installing data values intrinsically needs to be trusted. Of course, to achieve this advantage, there must exist effective kernel supports. It is true that the values stored as access control data, the information used by the system to determine which users may access each data item, must be assured to be exactly correct. However, since this data collection is very small compared to the rest of the database, and mostly specialized "retrieval requests" are made of it (i.e. those needed to check an intended data access), the access methods can be quite simple.

The basic approach depends on the fact that it is possible to structure a database management system so that a small amount of isolated software has the following

responsibilities:

1. assure that a given data item is stored with the correct name labelling it,
2. check the access control constraints on each application data access request, and
3. deliver only the data items contained in the named objects.

This small amount of software composes the data management kernel. The structuring is accomplished in the following way. The actual software security base is composed of two groups of modules, one similar in structure and function to an operating system kernel, and one which is associated with user level interactions. Reliable enforcement by the underlying operating system of separate domains, each containing portions of the rest of the data management system, is also needed. Otherwise, no special interaction between database management system security enforcement and the basic operating system protection facilities is needed.

To understand how these two modules operate with the major database software, consider how normal operation occurs; update first. The user inputs the update command, together with the logical name(s) of the entity or entities to be updated and their new values. A certified kernel input module makes a copy of each entity name - value pair, for reasons that will be pointed out below. The copies are saved in base kernel storage, together with the id of the user.

The update command plus the data is passed to the major data management module (DMM) which operates in a protection domain separate from the user's. There is one DMM, in a single domain, operating for all concurrent users of the data management system. The usual data management tables, including data descriptions, schemas and synchronization variables, are located there. The DMM is responsible for many of the standard data management functions, including logical to physical translation, synchronization and locking, audit, backup and recovery orders, generation of additional retrievals to support views, collection of performance data, and so forth. The code comprising the DMM is not trusted and need not be certified. The only exception occurs because certain confinement channels can be exercised by the DMM, so that it is assumed to contain no so called "trojan horses".

The output of the DMM is a request to the data management kernel to perform an I/O that will cause the actual update. The request is composed of a pair containing the logical name of the entity involved and the physical parameters necessary for the I/O, but not including the data itself. That information is supplied by an independent path. In particular, each logical name-value pair obtained by the kernel input module is passed to an uncertified

formatting module. Each such formatting module runs in its own newly created domain and processes a single entity value, using standard read only data description tables to prepare the physical representation of the data item for actual I/O. The item is matched by the base kernel with the rest of the I/O request from the DMM.

Using this segmented architecture, the physical data value about to be written can depend only on the value of the logical data entered by the user or by his program (or on tables of constants) even though most of the software which has been involved is uncertified.

The base kernel must now insure that the physical location about to be written properly corresponds to the logical name by which a subsequent retrieval request would be issued. A simple way to do so would be to check a physical/logical map, which listed for each physical storage unit the name of the logical entity to which the storage corresponded. If the logical name presented by the DMM matches the tag on the physical location given by the DMM and the recorded access control data permits the operation, the kernel issues the I/O. The update has been performed in a certified fashion. (The actual redundant map is maintained in a slightly more complex way in the implementation.)

Retrieval in this architecture operates in a simpler fashion than update. The kernel input module is not involved in retrieval operations because no alteration of the data base occurs. The base kernel, upon receiving the request for retrieval from the DMM, merely checks the physical/logical map and the stored access control data to determine whether the retrievals being attempted are permitted. The base kernel then issues the I/O with the output going into the user data area. Formatting software running in the user domain alters the data to match the user's schema and then the results are returned to the user.

That database kernel structure just outlined has been implemented. First, a prototype kernel was constructed which controlled read access in an effective way. Update access controls were designed and implemented, but not fully tested. These steps were reported in a doctoral dissertation [DOWN79].

While the doctoral work and its "retrieve/update kernel" showed that data management security using kernel technology was promising, it was not conclusive. That was because any commercially viable database system includes many other functions which affect the values of data and the mapping between names and actual data items. In fact, it is the complexity of this mapping that is a significant distinguishing factor between the data management security problem

and the same concerns in operating systems. Many of these additional functions are utilities that support database reorganization, index construction and deletion, data creation, etc.

Therefore, a subsequent effort was made to extend the basic kernel approach to include control of all these additional functions. Surprisingly, it was found that substantially all of them could be reformulated in terms of updates and retrievals, which were already handled. This reformulation did not lead to a serious, intrinsic loss of performance. The only exceptions found to this straightforward approach concerned certain critical system data structures, especially those which described the structure of the actual stored data themselves. It is common in database systems to treat these descriptive structures (the so called "data dictionary" in commercial terms) as normal stored data, so that all of the system's access methods can be directly applied. However, since the descriptions are an integral part of security enforcement, they must be treated specially by the database kernel. Fortunately, they are small and their use is highly stylized, so that a simple, special cased implementation can be provided. The resulting software must also be included in the full database kernel.

It is quite pleasing that even after the basic database kernel concept is extended to support management of access control information as well as utility operation, the resulting kernel is still strikingly small and simply structured, especially when compared to the dbms itself. For Ingres, the basic system is composed of approximately 250K bytes of code, while the kernel is between one and two orders of magnitude less.

Despite this success, one should not conclude that existing data management systems can necessarily be retrofitted in a manner similar to our experience with Ingres. That case study happened to exhibit a structure quite amenable to our needs. There is no particular reason to expect that other existing systems will also. Therefore, we believe that it is important that this kernel based dbms architecture be taken into account during the design and construction of data management software.

1.2 Network Security

In the previous contract, an approach to providing general network security was developed [POPE77b]. That approach effectively encapsulates the network protocol software from the operating system. In that way, the security kernel could be enhanced to provide encryption based network security without requiring the entire protocol software to be trusted. In the current contract period, that work was completed. Integration with the Data Secure Unix prototype was also done.

As part of that effort, we performed a general study of encryption and computer networks [POPE79b]. A detailed comparison and evaluation of public key and conventional encryption algorithms was performed from the viewpoint of integration into software systems. Despite the fact that public key systems have additional features beyond those exhibited by conventional encryption, it was found that those features, at least for the functions currently desired in a secure network, are not of significant practical use once the mechanism is integrated into the systems within which it must operate. Those results were reported in the professional literature, as well as in a doctoral dissertation.

The most significant expected advantage of public key encryption over conventional systems concerned digital signatures. Public key systems seemed superior. We have shown that this advantage is largely illusory, because of a defect in all previous signature proposals that permit authors to disclaim authorship at any time. In particular, the problem occurs if the author has access to the information on which the validity of signatures is based. By releasing this information, the author effectively can claim that anyone could forge signatures, including those previously signed.

Once this problem is realized, it becomes immediately clear that some trusted mechanism is necessary to protect the information on which the validity of signatures is based. We call that trusted mechanism the network signature registry. Various designs are possible, especially when such considerations as network, host, and registry failures are considered.

During the current contract period, we have examined these issues in detail, and explored several registry models and digital signature protocols, ranging from fully centralized to fully distributed operation. Both public-key based and conventional based encryption methods have been considered in the solutions. Not surprisingly, here again the particular encryption method seems to make little difference in the final solution.

We have proposed what appears to be a superior digital signature system. It is based on conventional encryption and a small amount of trusted hardware and software, a signature kernel. During the current contract period, a prototype implementation of the digital signature protocol was implemented under Unix. Several minor errors in the protocol were uncovered in this process, and the result is a rather convincing demonstration of the viability of digital signatures. Also, an examination of digital signature protocols with a goal towards verifying the security provided

by these protocols has begun. That work has only reached a preliminary state, but appears quite promising.

At UCLA, our approach to security problems has repeatedly involved the concept of a convincing demonstration of the security of the system, usually via verification techniques. As with operating system and database security, here again we are applying that methodology.

1.3. Reliable Distributed Systems Architectures

Our last major activity has concerned achieving the potential that distributed systems have, via redundancy, for higher reliability than centralized architectures. While this potential is well known and recognized, achieving this goal in practice requires great care. It is quite common to design systems where the interconnection of components, especially at software levels, increases rather than decreases the probability of system failure.

At the same time, we recognized how difficult it is in practice to build software for a distributed environment, and therefore focussed on architectural approaches that would minimize those difficulties. Part of the initial work concentrated on determining what the actual sources of difficulty were.

To maintain reality in our work, we chose to design and build a prototype which incorporated many of the innovative concepts we were evolving.

Most existing distributed systems were constructed by making relatively minor modifications to adapt single machine systems to permit them to interact with other copies of themselves, or even with other systems. The basic structure and operational philosophy of those systems was invariably preserved. Our goal was to understand, given the freedom to start largely anew, what the structure of a distributed system ought to be. One fundamental assumption we made concerned the interconnection network; it was to be of high bandwidth, low delay, with a low error rate - so called "thick wire" networks, represented by such examples as the Ethernet. Broadcast capability was not assumed. We explicitly ruled out Arpanet or Telenet style networks because of their limited bandwidth and significant delay. Satellite based networks were not explicitly addressed because of their significant delay.

General applications were to be supported, with focus on "computer utility" functions and data management. High reliability/availability was essential, and ease of programming very important. We were willing to insist that all sites in the network run a given system, but the sites should be able to vary widely in power and storage capacity.

At the outset, good performance was considered to be an important measure of success.

An early version of that prototype distributed system, LOCUS, is now operational at UCLA [POPEB1]. LOCUS is a distributed operating system whose architecture strongly addresses the issues of network transparency, reliability and availability, and performance that drove the design. The machines in a LOCUS net cooperate to give all users the illusion of operating on a single machine; the network is essentially invisible. Nevertheless, each machine is a complete system and can operate gracefully alone. Data storage is designed to be automatically replicated to the degree indicated by associated reliability profiles. Graceful operation in the face of network partitions, as well as nodal failures, is supported. It is expected that these characteristics are suitable for the support of a wide variety of applications, including general distributed computing, office automation, and database management.

The existing system especially incorporates our early ideas regarding reliability, environment, and performance. The specific accomplishments in each of these areas are discussed below. It should be realized that these concepts are still evolving and their refinement is an important component of our future work.

1.3.1. Network Transparency

As real distributed systems come into existence, an unpleasant truth has been learned: the development of software for distributed systems, and for true distributed applications, is often far harder to design, implement, debug, and maintain than the analogous software written for a centralized system. The reasons include a much richer set of error and failure modes to deal with, as well as the lack of a consistent interface across machines by which both local and remote resources can be accessed. An example of a richer error mode is the reality of partial failures; portions of a distributed computation may fail while the others continue unaware. In a centralized system, one typically assumes that the entire computation stops.

Further, local storage may be limited, necessitating that the user explicitly move copies of files around the network, archiving and garbage collecting his own storage. Redundant copies for the sake of reliability is the user's concern. Keeping track of different versions of what is intended to be the same file requires user attention, especially when the copies have resulted from network partitions (leading to parallel changes). As a result, the application program and user must explicitly deal with each of these facts, at considerable cost in additional software. On a centralized machine, with a single integrated file system,

many of these problems do not exist, or are more gracefully handled.

An appealing solution to this increasingly serious problem is to develop a network operating system that supports a high degree of network transparency; all resources are accessed in the same manner independent of their location. If open (file-name) is used to access local files, it also is used to access remote files. That is, the network becomes "invisible", in a similar manner to the way that virtual memory hides secondary store. Of course, one still needs some way to control resource location for optimization purposes, but that control should be separated from the syntax and semantics of the system calls used to access the resources. Ideally then, one would like the graceful behavior of an integrated storage system for the entire network while still retaining the many advantages of the distributed system architecture. The existence of the network should not concern the user or application programs in the way that resources are accessed.

There are also some system aspects that militate against full transparency. If the hardware bases of each site aren't the same, then it may be necessary to have different load modules correspond to a given name, so that when a user (or another program) issues a standard name, the appropriate file gets invoked as a function of the machine on which the operation is to be performed. There are other examples as well; they all have the characteristic that a standard name is desired for a function or object that is replicated at some or all sites, and a reference to that name needs to be mapped to the local, or nearest instance in normal circumstances.

Nevertheless, in other circumstances, a globally unique name for each instance is also necessary; to install software, do system maintenance functions, etc. A solution that preserves network transparency and provides globally unique names within the normal name space while still supporting site dependent mapping for these special cases is needed. Our short term solution to this problem involves appending tags to file names which by convention indicate site dependent information. Various system and user call options default the tag value appropriately.

1.3.2. Reliability

Reliability and availability represent a whole other aspect of distributed systems which has had considerable impact on LOCUS. Four major classes of steps have been taken to achieve the potential for very high reliability and availability present in distributed systems with redundancy.

First, an important aspect of reliable operation is the

ability to substitute alternate versions of resources when the original is found to be flawed. In order to make the act of substitution as straightforward as possible, it is desirable for the interfaces to the resource versions to look identical to the user of those resources. While this observation may seem obvious, especially at the hardware level, it also applies at various levels in software, and is another powerful justification for network transparency. In LOCUS, copies of a file can be substituted for one another with no visibility to application code.

From another point of view, once a significant degree of network transparency is present, one has the opportunity to enhance system reliability by substituting software as well as hardware resources when errors are detected. In LOCUS, considerable advantage is taken of this approach. Since the file system supports automatic replication of files transparently to application code, it is possible for graceful operation in the face of network partition to take place. If the resources for an operation are available in a given partition, then the operation may proceed, even if some of those are data resources replicated in other partitions. A partition merge procedure detects any inconsistencies which may result from this philosophy, and for those objects whose semantics the system understands (like file directories, mailboxes, and the like), automatic reconciliation is done.

Second, the concept of committing a file is supported in LOCUS. For a given file, one can be assured that either all the updates are done, or none of them are done. Commit normally occurs automatically when a file is closed if the file had been open for write, but application software may request commits at any time during the period when a file is open.

Third, even though a high level of network transparency is present in the syntax and semantics of the system interface, each site is still largely autonomous. For example, when a site is disconnected from the network, it can still go forward with work local to it.

Fourth, the interaction among machines is strongly stylized to promote "arms length" cooperation. The nature of the low level interfaces and protocols among the machines permits each machine to perform a fair amount of defensive consistency checking of system information. As much as feasible, maintenance of internal consistency at any given site does not depend on the correct behavior of other sites. (There are, of course, limits to how well this goal can be achieved.) Each site is master of its own resources, so it can prevent flooding from the network.

1.3.3. Recovery

Given that a data resource can be replicated, a policy issue arises. When the network is partitioned and a copy of that resource is found in more than one partition, can the resource be modified in the various partitions? It was felt important for the sake of availability that such updates be permitted. Of course, this freedom can easily lead to consistency conflicts when the partitions are merged. However, our view is that such conflicts are likely to be rare, since actual sharing in computer utilities is known to be relatively low.

Further, we developed a simple, elegant algorithm to detect conflicts if they have occurred [PARK81]. The core of the method is to keep a version vector with each copy of the data object. There are as many elements in the vector as there are sites storing the object. Whenever an update is made to a copy of the object at a given site, that site's element of the version vector associated with the updated copy is incremented. The conflict detection criterion is then very simple. When merging two copies of an object, compare their version vectors. If one dominates the other (i.e. each element is pairwise greater than or equal to its corresponding element), then there is no conflict; the copy associated with the dominating vector should propagate. Otherwise a conflict exists.

Most significant, for those data items whose update and use semantics are simple and well understood, it may be quite possible to reconcile the conflicting versions automatically, in a manner that does not have a "domino effect"; i.e. such a reconciliation does not require any actions to data items that were updated during partitioned operation as a function of the item(s) being reconciled [FAIS81].

Good examples of data types whose operations permit automatic reconciliation are file directories and user mailboxes. The operations which apply to these data types are basically simple: add and remove. The reconciled version is the union of the several versions, less those items which have been removed. There are of course situations where the system does not understand the semantics of the object in conflict. The LOCUS philosophy is to report the conflict to the next level of software, in case resolution can be done there. An example of this case might be d.a management software. Eventually, the conflict is reported to the user.

1.3.4. Performance and its Impact on Software Architecture

"In software, virtually anything is possible; however, few things are feasible." [Cheatham] While the goals outlined in the preceding sections may be attainable in principle, the more difficult goal is to meet all of the

above while still maintaining good performance within the framework of a well structured system without a great deal of code. A considerable amount of the LOCUS design was tempered by the desire to maintain high performance. Perhaps the most significant design decisions in this respect are:

1. specialized "problem oriented protocols",
2. integrated rather than partitioned function, and
3. special handling for local operation.

Below, we discuss each of these in turn.

1.3.4.1. Problem Oriented Protocols

It is often argued that network software should be structured into a number of layers, each one implementing a protocol or function using the characteristics of the lower layer. In this way the difficulties of building complex network software are eased; each layer hides more and more of the network complexities and provides additional function. Thus layers of "abstract networks" are constructed. More recently, however, it has been observed that layers of protocol generally lead to layers of performance cost. In the case of local networks, it is common to observe a cost of up to 5,000 instructions being executed to move a small collection of data from one user program out to the network.

In a local network, we argue that the approach of layered protocols is frequently wrong, at least as it has been applied in long haul nets. Functionally, the various layers were typically dealing with issues such as error handling, congestion, flow control, name management, etc. In our case, these functions are not very useful, especially given that they have significant cost. By careful design, one can build special case solutions which integrate these issues with the higher level functions they support.

These observations lead us to develop specialized problem oriented protocols for the problem at hand. In Locus, for example, when a user wishes to read a page of a file, the only message that is sent from the using site to the storage site is a read message request. A read is one of the primitive, lowest level message types. There is no connection management, no acknowledgement overhead, etc. The only software ack in this case is the delivery of the requested page.

Our experience with these lean, problem oriented protocols has been excellent. The effect on system performance has been dramatic, as pointed out below.

1.3.4.2. Functional Partitioning

It has become common in some local network developments to rely heavily on the idea of "servers", where a particular

machine is given a single role, such as file storage, name lookup, authentication or computation. We call this approach the server model of distributed systems. Thus one speaks of "file servers", "authentication servers", etc. However, to follow this approach purely is inadvisable, for several reasons. First, it means that the reliability/availability of an operation which depends on multiple servers is the product of the reliability of all the machines and network links involved. The server design insures that, for many operations of interest, there will be a number of machines whose involvement is essential.

Second, because certain operations involve multiple servers, it is necessary for multiple machine boundaries to be crossed in the midst of performing the operation. Even though the cost of network use has been minimized in LOCUS as discussed above, it is still far from free; the cost of a remote procedure call or message is still far greater than a local procedure call. One wants a design where there is freedom to configure functions on a single machine when the situation warrants. Otherwise serious performance costs may result, even though the original designers believed their functional decomposition was excellent.

Third, since it is unreasonable to follow the server machine philosophy strictly, one is led to multiple implementations of similar functions; to avoid serious performance costs, a local cache of information otherwise supplied by a server is usually provided for at least some server functions. The common example is file storage. Even though there may be several file servers on the network, each machine typically has its own local file system. It would be desirable to avoid these additional implementations if possible.

An alternative to the server model is to design each machine's software as a complete facility, with a general file system, name interpretation mechanism, etc. Each machine in the local network would run the same software, so that there would be only one implementation. Of course, the system would be highly configurable, so that adaptation to the nature of the supporting hardware as well as the characteristics of use could be made. We call this view the integrated model of distributed systems. LOCUS takes this approach.

1.3.4.3. Local Operation

The site at which the file access is made, (the Using Site, US), may or may not be the same as the site where the file is stored (the Storage Site, SS) or where file synchronization is performed (the Current Synchronization Site, CSS). In fact, any combination of these roles are possible, or all may be played by different sites. When multiple roles

are being played by a single site, it is important to avoid much of the mechanism needed to support full, distributed operation. That is, when operating essentially locally, performance costs should not increase because of the mechanisms for the general case.

These optimizations are supported in LOCUS. For example, if $CSS = SS = US$ for a given file open, then this fact is detected immediately and virtually all the network support overhead is avoided. The cost of this approach is some additional complexity in protocols and system nucleus code.

The system design is intended to support machines of heterogeneous power interacting in an efficient way; large mainframes and small personal computers sharing a replicated file system, for example. Therefore, when a file is updated, it is not desirable for the requesting site to wait until copies of the update have propagated to all sites storing copies of the file, even if a commit operation is desired. The design choice made in LOCUS is for updated pages to be posted, as they are produced, at the storage site providing the service. When the file is closed, the disk image at the storage site is updated and the using site program now continues. Copies of the file are propagated to all other storage sites for that file in parallel.

1.3.5. Performance Results

To satisfactorily evaluate the performance of a network transparent system such as LOCUS, one would like answers to at least the following questions. First, when all the resources involved in an operation are local, what is the cost of that operation, especially compared to the corresponding system that does not provide for network operation, either in its design or implementation? This is often a difficult comparison to make. It is not sufficient just to "turn off" the network support code in LOCUS, as the existence of that code altered the very structure of the system in ways that wouldn't happen otherwise. Fortunately, in our case, we could compare local operation under LOCUS with the corresponding situation under standard Unix.

The performance of remote access is also of param. int importance, since, after all, it is distributed operation that largely motivated the LOCUS architecture.

Three initial experiments are reported here:

- a) reading one file page,
- b) writing one file page, and
- c) sequentially reading a 600 block file.

Each experiment was run on:

- a) standard Version 7 Unix,
- b) local LOCUS (i.e. the program and data both resided at the same machine), and
- c) distributed LOCUS (i.e. the data resided on the machine used in cases a and b, but the program ran on another machine.)

For the first two experiments, the quantity reported is system time - the amount of cpu time consumed in servicing the request. For distributed LOCUS, the quantity given is the sum of the system time at the using site and the storage site. In the third experiment, total elapsed time is reported.

A great deal of care was taken to assure comparability of results; the same data blocks were read, the same numbers of buffers were used, the same disk was employed, etc. The tests were repeated multiple times. The values reported are means, but the standard deviations were very small. The machines on which these measurements were run are DEC PDP-11/45s, with an average time of greater than 2 microseconds per instruction. The disk used is capable of delivering one data page per 15 milliseconds if there is no head movement.

System Time Measurements
(one data-page access - results in milliseconds)

System	Read	Write
Unix	6.07	4.28
Local LOCUS	6.15	4.27
Dist. LOCUS	14.27	7.50

The distributed LOCUS results are further decomposed into the load at the using site and the storage site. These results are as follows.

Distributed LOCUS System Loads

Test	Using Site	Storage Site
Read Page	6.30	7.97
Write Page	4.27	3.23

The sequential activity results are as follows.

Elapsed Time Measurements
(600 block sequential read)

System	Time (seconds)	Milliseconds/page
Unix	9.40	15.6
Local LOCUS	9.15	15.25
Dist. LOCUS	10.31	17.18

1.3.6. Interpretation of Results

In our view, these results strongly support the argument for network transparency. Local operation for the cases measured is clearly comparable to a system without such a facility. There is more system overhead for remote operation, but this is not surprising. Two network interface devices are involved in every remote access in addition to the one storage device. For a read, there are two messages involved, the request and the data-reply; hence a total of four additional I/Os must occur beyond the single disk I/O that was always present. That additional cost is about 4000 instructions in LOCUS (8 milliseconds, 2 instructions/millisecond). In the absence of special hardware or microcode to support network I/O, we consider this result quite reasonable.

It is especially encouraging that for operations which can take advantage of the involvement of multiple processors, the speed of remote transparent access is indeed comparable to local access. Sequential processing is one such example, since prefetching of data, both from the disk and across the network, is supported in LOCUS. Here, access in all cases is running at the maximum limit of the storage medium. No significant delay is imposed by the network transparent system.

Most important, one should compare the performance of a distributed system such as LOCUS, where the network support is integrated deep into the software system architecture, with alternate ways of gaining access to remote resources. The traditional means of layering software on top of centralized systems leads to dramatically greater overhead. Before the development of LOCUS, Arpanet protocols were run on the same network hardware connecting the PDP-11s. Throughput was not even within an order of magnitude of the results reported here.

Several caveats are in order however as these results are examined. First, at the time these measurements were made, replicated storage had not yet been implemented. Hence no conclusions in that respect can be made. Second, because of the lack of available network interconnection hardware, only a small network configuration is available for study, and so only light loading occurs. On the other hand, since most computer utility experience shows limited

concurrent sharing, we expect little synchronization interference when the network is significantly larger and more users are supported.

LOCUS also requires more memory to operate than standard Unix for comparable performance. This fact occurs for two principal reasons. First, there is more code in the LOCUS nucleus than in the Unix kernel, in part because of low level network support and synchronization mechanisms, because of more sophisticated buffer management, and also because of all the server process code. Second, LOCUS data structures are generally larger than the corresponding structures in Unix. For example, a LOCUS file descriptor has additional fields for a version vector, site information, and (when at a storage site) a bit map marking which sites are currently being served (i.e. have this file open). For the configuration at UCLA, Unix requires 41K bytes code and 44K bytes data. The equivalent LOCUS system (with the same number of buffers, etc.) would require 53K bytes code and 51K bytes data.

LOCUS also contains considerable additional code to handle recovery, but this code is run as a conventional user process, and therefore is not locked down.

1.4. Summary of Results

During the current contract, the work done can be quickly summarized as:

1. The Data Secure Unix operating system was completed, including documentation, verification efforts, and transfer of technology.
2. General principles of secure operation in a network environment were developed, especially with respect to the integration of encryption into systems architectures. Digital signature protocols were investigated, and the functionality of differing encryption methods were compared.
3. Extensive efforts were devoted to an unconstrained examination of software architectures for local area networks. A prototype system, LOCUS, was built to test and refine these ideas.

REFERENCES

DOWN77 Downs, D., and Popek, G. J., "A kernel design for a secure data base management system," Proceedings of the Third International Conference on Very Large Data Bases, Tokyo, Japan, October, 1977, 507-514.

DCWN79 Downs, D., and Popek, G., "Data base system security and Ingres," Proceedings of the conference on Very Large Data Bases, 1979, Rio De Janeiro.

KAMP77 Kampe, M., Kline, C. S., Popek, G. J., and Walton, E. J., The UCLA Data Secure Operating System Prototype, Computer Science Department, University of California, Los Angeles, Technical Report 77-3, July 1977.

KEMM78 Kemmerer, R. A., A proposal for the formal verification of the security properties of the UCLA Secure UNIX Operating System Kernel, Computer Science Department, University of California, Los Angeles, Technical Report 78-1 (UCLA-ENG-7810), February 1978.

KEMM79 Kemmerer, R. A., Formal verification of the UCLA Security Kernel: Abstract model, mapping functions, theorem generation, and proofs, Ph.D. thesis, UCLA-ENG-7956, University of California at Los Angeles, 1979.

KLIN77 Kline, C. S., and Popek, G. J., Encryption in computer network security, Computer Science Department, University of California, Los Angeles, Technical Report 77-2, April 1977.

KLIN79 Kline, C. S., and Popek, G. J., "Public key vs. conventional key encryption", Proceedings of the National Computer Conference, 48 (1979), AFIPS Press, Arlington, Va., 831-837.

MENA79 Menasce, D. A., Rudisin, G. J., Popek, G. J., and Kline, C. S., A proposed architecture for the distributed secure system base, Computer Science Department, University of California, Los Angeles, Technical Report 79-10 (UCLA-ENG-7957), September 1979.

POPE73 Popek, G. J., "Correctness in access control," Proceedings of the ACM National Conference, Atlanta, Georgia, 1973, 236-241.

POPE74a Popek, G. J., "Protection structures," IEEE Computer, (Jul. 1974), 22-33.

POPE74b Popek, G. J., "A principle of kernel design," Proceedings of the National Computer Conference, 43 (1974), AFIPS Press, Arlington, Va., 977-978.

POPE74c Popek, G. J., and Kline, C. S., "Verifiable secure operating system software," Proceedings of the National Computer Conference, 43 (1974), AFIPS Press, Arlington, Va., 145-151.

POPE74d Popek, G. J., and Kline, C. S., "The design of a verified protection system," Proceedings of the IRIA International Workshop on Protection in Operating Systems, Rocquencourt, France, August, 1974, 183-196.

POPE75a Popek, G. J., and Kline, C. S., "A verifiable protection system," Proceedings of the 1975 International Conference on Reliable Software, Los Angeles, Ca., April 21-23, 1975, 294-304.

POPE75b Popek, G. J., and Kline, C. S., "The PDP-11 virtual machine architecture: a case study," Proceedings of the Fifth Symposium on Operating Systems Principles, Austin, Texas, October 1975.

POPE76 Popek, G. J., and Farber, D. A., "On computer security verification," Twelfth IEEE Computer Society International Conference: Compcon 76, San Francisco, Ca., Feb. 1976, 140-145.

POPE77a Popek, G. J., Horning, J. J., Lampson, B. W., Mitchell, J. G., and London, R. L., "Notes on the design of EUCLID," Proceedings of an ACM Conference on Language Design for Reliable Software, Raleigh, North Carolina, March, 1977, 11-18. Also in SIGPLAN Notices, 12, 3 (Sept. 1977).

POPE77b Popek, G. J., and Kline, C. S., "Encryption protocols, public key algorithms and digital signatures in computer networks," Foundations of Secure Computing, R. DeMillo, et. al., eds., Academic Press, New York, 1978, 133-153.

POPE78a Popek, G. J., and Kline, C. S., "Design issues for secure computer networks", Operating Systems, An Advanced Course, R. Bayer, R. M. Graham, G. Seegmuller, Eds., Springer-Verlag, New York, 1978

POPE78b Popek, G. J., and Farber, D. A., "A model for verification of data security in operating systems," Communications of the ACM, 21, 9 (Sept. 1978), 737-749.

POPE78c Popek, G. J., and Kline, C. S., "Issues in kernel design," Proceedings of the National Computer Conference, 47 (1978), AFIPS Press, Arlington, Va., 1079-1086.

POPE78d Popek, G. J., Kampe, M., Kline, C. S., Stoughton, A. H., Urban, M. P., and Walton, E. J., UCLA Data Secure Unix -- a secure operating system: software architecture, Technical Report 78-7 (UCLA-ENG-7854), Computer Science Department, University of California, Los Angeles, 1978.

POPE79 Popek, G. J., Kampe, M., Kline, C. S., Stoughton, A., Urban, M., and Walton, E. J., "UCLA Secure Unix," Proceedings of the National Computer Conference, 48 (1979), AFIPS Press, Arlington, Va., 355-364.

URBA79 Urban, M. P., The design and implementation of a file policy manager for the UCLA Data Secure Unix System, M.S. Thesis, Computer Science Department, University of California, Los Angeles, 1979.

WALK77 Walker, B. J., Verification of the UCLA Security Kernel: Data Defined Specifications, M. S. Thesis, Computer Science Department, University of California, Los Angeles, 1977.

WALK80 Walker, B. J., Kemmerer, R. A., and Popek, G. J., "Specification and verification of the UCLA Unix Security Kernel," Communications of the ACM, 23, 2 (Feb. 1980), 118-131.

WALT75 Walton, E. J., The UCLA Security Kernel, M.S. Thesis, Computer Science Department, University of California, Los Angeles, June 1973.

BIBLIOGRAPHY

Popek, G. J. "Correctness in Access Control," Proceedings of the ACM National Conference, Atlanta, Georgia, August 1973, pp. 236-241.

Popek, G. J. and R. P. Goldberg. "Formal Requirements for Virtualizable Third Generation Architectures," ACM/SIGOPS Fourth Symposium on Operating System Principles, Yorktown Heights, New York, October 15-17, 1973. Revised version published in Communications of the ACM, 17(7):412-421, July 1974.

Popek, G. J. "A Principle of Kernel Design," AFIPS Conference Proceedings, Vol. 43: 1974 National Computer Conference, Chicago, Illinois, May 6-10, 1974, AFIPS Press, Montvale, New Jersey, 1974, pp. 977-978.

Popek, G. J. and C. S. Kline. "Verifiable Secure Operating System Software," AFIPS Conference Proceedings, Vol. 43: 1974 National Computer Conference, Chicago, Illinois, May 6-10, 1974, AFIPS Press, Montvale, New Jersey, 1974, pp. 145-151.

Popek, G. J. "Protection Structures," IEEE Computer, June 1974, pp. 22-23.

Popek, G. J. "A Note on Secure Data Base Design," Computer Science Department, University of California, Los Angeles, Technical Report 74-4, July 1974.

Popek, G. J. and C. S. Kline. "The Design of a Verified Protection System," Proceedings of the IRIA International Workshop on Protection in Operating Systems, Rocquencourt, France, August 13-14, 1974, pp. 183-196.

Snuggs, M. A. L., G. J. Popek, and R. J. Peterson. "Data Base System Objectives as Design Constraints," Proceedings of the ACM National Conference, San Diego, California, November 1974, pp. 641-647.

Popek, G. J. "On Data Secure Computer Networks," Proceedings of the ACM SIGCOMM/SIGARCH Workshop on Network Communications, Santa Monica, California, March 1975, pp. 59-62.

Popek, G. J. and C. S. Kline. "A Verifiable Protection System," Proceedings of the 1975 International Conference on Reliable Software, Los Angeles, California, April 21-23, 1975, pp. 294-304.

Walton, E. J. The UCLA Security Kernel, M.S. in Computer Science, Computer Science Department, University of California, Los Angeles, June 1975.

Popek, G. J. "On the Current State of Protection in Computer Systems," Eleventh IEEE Computer Society Conference: Compcon '75 Fall, Washington, D.C., September 9-11, 1975, pp. 40-41.

Popek, G. J. and C. S. Kline. "The PDP-11 Virtual Machine Architecture: A Case Study," Proceedings of the Fifth Symposium on Operating Systems Principles, Austin, Texas, October 1975.

Walton, E. J. "The UCLA Pascal Translation System," Computer Science Department, University of California, Los Angeles, UCLA-ENG-7954 (DAHC-73-C-0368), January 1976.

Popek, G. J. and D. A. Farber. "On Computer Security Verification," Twelfth IEEE Computer Society International Conference: Compcon '76 Spring, San Francisco, California, February 24-26, 1976, pp. 140-145.

Farber, D. A. "A Model of Program Translation," Computer Science Department, University of California, Los Angeles, Technical Report 76-3, August 1976.

Popek, G. J., J. J. Horning, B. W. Lampson, R. L. London, and J. G. Mitchell. "The Programming Language Euclid," Computer Science Department, University of California, Los Angeles, Technical Report 76-4, September 1976.

Walker, B. J. "Note on Proof of Concrete Code," Computer Science Department, University of California, Los Angeles, Technical Report 76-5, September 1976.

Abraham, S. M. A Protection Design for the UCLA Security Kernel, M.S. in Computer Science, Computer Science Department, University of California, Los Angeles, December 1976.

Popek, G. J., J. J. Horning, B. W. Lampson, J. G. Mitchell, and R. L. London. "Notes on the Design of EUCLID," Proceedings of an ACM Conference on Language Design for Reliable Software, Raleigh, North Carolina, March 28-30, 1977, ACM, Inc., New York, New York, 1977, pp. 11-18. Also published in SIGPLAN Notices, 12(3):11-18, 1977.

Kline, C. S. and G. J. Popek. "Encryption in Computer Network Security," Computer Science Department, University of California, Los Angeles, Technical Report 77-2, April 1977.

Kampe, M., C. S. Kline, G. J. Popek, and E. J. Walton. "The UCLA Data Secure Operating System Prototype," Computer Science Department, University of California, Los Angeles, Technical Report 77-3, July 1977.

Popek, G. J. and C. S. Kline. "Design Issues for Secure Computer Networks," presented at the Advanced Course in Operating Systems, July 28-August 5, 1977 and March 29-April 6, 1978, Munich, Germany. In: Operating Systems, An Advanced Course, edited by R. Bayer et al., Springer-Verlag, Berlin, Germany, 1978, pp. 518-546 (Lecture Notes in Computer Science, Vol. 60, edited by G. Goos and J. Hartmanis).

Popek, G. J. and C. S. Kline. "Issues in Kernel Design," presented at the Advanced Course in Operating Systems, July 28-August 5, 1977 and March 29-April 6, 1978, Munich, Germany. In: Operating Systems, An Advanced Course, edited by R. Bayer et al., Springer-Verlag, Berlin, Germany, 1978, pp. 209-226 (Lecture Notes in Computer Science, Vol. 60, edited by G. Goos and J. Hartmanis). Also published in AFIPS Conference Proceedings, Vol. 47: 1978 National Computer Conference, Anaheim, California, June 5-8, 1978, AFIPS Press, Montvale, New Jersey, 1978, pp. 1079-1086.

Menasce, D. A., G. J. Popek, and R. R. Muntz. "A Locking Protocol for Resource Coordination in Distributed Systems," Computer Science Department, University of California, Los Angeles, Technical Report 77-6 (UCLA-ENG-7808), October 1977.

Popek, G. J. and C. S. Kline. "Encryption Protocols, Public Key Algorithms and Digital Signatures in Computer Networks," Proceedings of the Atlanta Conference on Fundamentals of Secure Computing, Atlanta, Georgia, October 2-5, 1977. Also published in Foundations of Secure Computation, edited by R. A. de Millo, et al., Academic Press, Inc., New York City, New York, 1978, pp. 133-153.

Downs, D. and G. J. Popek. "A Kernel Design for a Secure Data Base Management System," Proceedings of the Third International Conference on Very Large Data Bases, Tokyo, Japan, October 6-8, 1977, IEEE, 1977, pp. 507-514. Also published in Data Base Engineering, 1(4):8-15, December 1977.

Walker, B. J. "Verification of the UCLA Security Kernel: Data Defined Specifications," Computer Science Department, University of California, Los Angeles, Technical Report 77-9 (UCLA-ENG-7809), November 1977 (Also published as a M.S. Thesis).

Kemmerer, R. A. "A Proposal for the Formal Verification of the Security Properties of the UCLA Secure UNIX Operating System Kernel," Computer Science Department, University of California, Los Angeles, Technical Report 78-1 (UCLA-ENG-7810), February 1978.

Popek, G. J. "Secure Distributed Processing Systems: Quarterly Management Report," Computer Science Department, University of California, Los Angeles, Technical Report 78-2, April 30, 1978.

Menasce, D.A., G.J. Popek, and R.R. Muntz. "A Locking Protocol for Resource Coordination in Distributed Databases," Proceedings of the 1978 ACM/SIGMOD International Conference on Management of Data, Austin, Texas, May 31-June 2, 1978 (extended abstract). Full paper published in Proceedings of the Third Berkeley Workshop on Distributed Data Management and Computer Networks, Berkeley, California, August 29-31, 1978. Also published in ACM Transactions on Database Systems, 5(2):103-138, June 1980.

Popek, G.J. "Security in Network Operating Systems: A Survey," report prepared for the Institute for Computer Sciences and Technology, National Bureau of Standards, June 30, 1978.

Popek, G.J. "Secure Distributed Processing Systems: Quarterly Technical Report," Computer Science Department, University of California, Los Angeles, Technical Report 78-5 (UCLA-ENG-7853), June 1978.

Badal, D.Z. and G.J. Popek. "A Proposal for Distributed Concurrency Control for Partially Replicated Distributed Database Systems," Proceedings of the Third Berkeley Workshop on Distributed Data Management and Computer Networks, Berkeley, California, August 29-31, 1978, pp. 273-286.

Menasce, D.A. and R.R. Muntz. "Locking and Deadlock Detection in Distributed Databases," Proceedings of the Third Berkeley Workshop on Distributed Data Management and Computer Networks, Berkeley, California, August 29-31, 1978, pp. 215-232. Also published in IEEE Transactions on Software Engineering, 5(3):195-202, May 1979.

Popek, G.J., M. Kampe, C.S. Kline, A.H. Stoughton, M.P. Urban, and E.J. Walton. "UCLA Data Secure UNIX - A Secure Operating System: Software Architecture," Computer Science Department, University of California, Los Angeles, Technical Report 78-7 (UCLA-ENG-7854), August 1978.

Popek, G.J. and D.A. Farber. "A Model for Verification of Data Security in Operating Systems," Communications of the ACM, 21(9):737-749, September 1978.

Menasce, D.A., G.J. Popek, and R.R. Muntz. "Centralized and Hierarchical Locking in Distributed Databases," IEEE Computer Society International Computer Software and Applications Conference: Distributed Data Base Management (Tutorial), Chicago, Illinois, November 1978, IEEE, New York City, New York, 1978, pp. 178-195.

Popek, G. J. "Secure Distributed Processing Systems: Quarterly Technical Report," Computer Science Department, University of California, Los Angeles, Technical Report 78-13 (UCLA-ENG-7955), December 1978.

Menasce, D. A., R. R. Muntz, and G. J. Popek. "A Formal Model of Crash Recovery in Computer Systems," Proceedings of the Twelfth Hawaii International Conference on System Sciences, Vol. I: Selected Papers in Software Engineering and Mini-Micro Systems, Honolulu, Hawaii, January 4-5, 1979, Western Periodicals Company, New York, 1979, pp. 28-35.

Chiappa, N. and A. H. Stoughton. "Programming the Local Network Interface," MIT Laboratory for Computer Science, Network Implementation Note No. 2, January 12, 1979.

Urban, M. P. The Design and Implementation of a File Policy Manager for the UCLA Data Secure Unix System, M.S. in Computer Science, Computer Science Department, University of California, Los Angeles, March 1979.

Badal, D. Z. and G. J. Popek. "Cost and Performance Analysis of Semantic Integrity Validation Methods," Proceedings of the 1979 ACM/SIGMOD International Conference on Management of Data, Boston, Massachusetts, May 30-June 1, 1979.

Kline, C. S. and G. J. Popek. "Public Key vs. Conventional Key Encryption," AFIPS Conference Proceedings, Vol. 48: 1979 National Computer Conference, New York City, New York, June 4-7, 1979, AFIPS Press, Montvale, New Jersey, 1979, pp. 831-837.

Popek, G. J., M. Kampe, C. S. Kline, A. H. Stoughton, M. P. Urban, and E. J. Walton. "UCLA Secure UNIX," AFIPS Conference Proceedings, Vol. 48: 1979 National Computer Conference, New York City, New York, June 4-7, 1979, AFIPS Press, Montvale, New Jersey, 1979, pp. 355-364.

Badal, D. Z. Semantic Integrity, Consistency and Concurrency Control in Distributed Databases, Ph.D. in Computer Science, Computer Science Department, University of California, Los Angeles, June 1979.

Popek, G. J. "Secure Distributed Processing Systems: Quarterly Management Report," Computer Science Department, University of California, Los Angeles, Technical Report 79-8, June 30, 1979.

Popek, G. J. "Secure Distributed Processing Systems: Quarterly Technical Report," Computer Science Department, University of California, Los Angeles, Technical Report 79-9 (UCLA-ENG-7956), June 30, 1979.

Badal, D. Z. "On Efficient Monitoring of Database Assertions in Distributed Databases," Proceedings of the Fourth Berkeley Conference on Distributed Data Management and Computer Networks, Berkeley, California, August 28-30, 1979, pp. 125-137.

Menasce, D. A., G. J. Rudisin, G. J. Popek, and C. S. Kline. "A Proposed Architecture for the Distributed Secure System Base," Computer Science Department, University of California, Los Angeles, Technical Report 79-10 (UCLA-ENG-7957), September 1979.

Downs, D. and G. J. Popek. "Data Base Management Systems Security and INGRES," Proceedings of the Fifth International Conference on Very Large Data Bases, Rio de Janeiro, Brazil, October 3-5, 1979, IEEE, 1979, pp. 280-290.

Walker, B. J., R. A. Kemmerer, and G. J. Popek. "Specification and Verification of the UCLA Security Kernel," Preprints of the Seventh Symposium on Operating Systems Principles, Pacific Grove, California, December 10-12, 1979, pp. 23-24 (extended abstract). Also published in Communications of the ACM, 23(2):118-131, February 1980.

Menasce, D. A. "Coordination in Distributed Systems: Concurrency, Crash Recovery and Database Synchronization," Computer Science Department, University of California, Los Angeles, Technical Report 79-14 (UCLA-ENG-7977), December 1979 (Also published as a Ph. D. Dissertation, December 1978).

Kemmerer, R. A. "Formal Verification of the UCLA Security Kernel: Abstract Model, Mapping Functions, Theorem Generation, and Proofs," Computer Science Department, University of California, Los Angeles, Technical Report 79-15 (UCLA-ENG-7978), December 1979 (Also published as a Ph. D. Dissertation, June 1979).

Popek, G. J. and C. S. Kline. "Encryption and Secure Computer Networks," ACM Computing Surveys, 11(4):331-356, December 1979.

Rudisin, G. J. Architectural Issues in a Reliable Distributed File System, Computer Science Department, University of California, Los Angeles, Technical Report 80-xx (UCLA-ENG-8014), April 1980 (Also published as a M. S. Thesis, March 1980).

Kline, C. S. Data Security: Operating Systems and Computer Networks, Computer Science Department, University of California, Los Angeles, Technical Report 80-2, October 1980 (Also published as a Ph. D. Dissertation, October 1980).

Parker, D. S., G. J. Popek, G. Rudisin, A. Stoughton, B. Walker, E.

Walton, J. Chow, D. Edwards, S. Kiser, and C. Kline. "Detection of Mutual Inconsistency in Distributed Systems," Proceedings of the Fifth Berkeley Workshop on Distributed Data Management and Computer Networks, Berkeley, California, February 3-5, 1981, pp. 172-184. Also accepted for publication in IEEE Transactions on Software Engineering.

Stoughton, A.H. "Access Flow: A Protection Model Which Integrates Access Control and Information Flow," submitted for publication to 1981 Symposium on Security and Privacy, Oakland, California, April 27-29, 1981.

Faissol, S.Z. Operation of Distributed Database Systems Under Network Partitions, Computer Science Department, University of California, Los Angeles, Technical Report 81-1, June 1981 (Also published as a Ph.D. Dissertation, June 1981).

Popek, G.J., B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel. "LOCUS: A Network Transparent, High Reliability Distributed System," submitted for publication to Proceedings of the Eighth Symposium on Operating Systems Principles, Pacific Grove, California, December 1981.

UNIVERSITY OF CALIFORNIA
Los Angeles

Data Security: Operating Systems and Computer Networks

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy
in Computer Science

by

Charles Steven Kline

1980

(C) Copyright by
Charles Steven Kline
1980

The dissertation of Charles Steven Kline is approved.

Richard R. Muntz
Richard R. Muntz

Leonard Kleinrock
Leonard Kleinrock

James Frane
James Frane

Peter Ladefoged
Peter Ladefoged

Gerald J. Popk
Gerald J. Popk, Committee Chair

University of California, Los Angeles

1980

This dissertation is dedicated to my wife, Lanaii, who has tolerated the many years it has taken, and to my friends, who will finally not be able to tease me about it anymore.

CONTENTS

Chapter 1 - Computer security.....	1
1.1 Introduction.....	1
1.2 System Security.....	2
1.2.1 Theft of Services.....	4
1.2.2 Denial of Service.....	5
1.2.3 Data Security.....	7
1.2.4 Confinement.....	8
1.2.5 System Correctness.....	9
1.3 Implementation.....	9
1.4 Data Security Policies.....	14
1.5 Certification.....	15
1.6 Network Security.....	16
1.7 Dissertation Structure.....	19
 Chapter 2 - Data Security.....	 20
2.1 Introduction.....	20
2.2 Processes.....	20
2.3 Information Flow.....	21
2.3.1 States and State Transitions.....	21
2.3.2 Read, Write, and Information Flow.....	22
2.4 Domains.....	24
2.5 Data Security Policies.....	25
2.6 Systems.....	26
2.7 Proper Execution.....	27
2.8 Data Security.....	29
2.9 Comparison of Models.....	29
2.10 Confinement.....	30
 Chapter 3 - The UCLA Data Secure Unix Operating System.....	 32
3.1 Introduction.....	32
3.2 System Structure.....	33
3.2.1 Security Kernel.....	34
3.2.2 File/Policy Manager (FPM).....	35
3.2.3 Initiator/Dialoguer.....	36
3.2.4 Scheduler.....	37
3.2.5 User Processes.....	38
3.3 Protected Objects.....	39
3.3.1 Processes.....	39
3.3.2 Pages.....	40
3.3.3 Devices.....	41
3.4 Kernel Specifications.....	42
3.4.1 Kernel Data Structures.....	42
3.4.2 Kernel Functions.....	43
3.4.3 Kernel Calls.....	46
3.4.4 Pseudo Interrupts.....	57
 Chapter 4 - Intuitive Verification.....	 58
4.1 Introduction.....	58

4.2	The Verification Methodology.....	59
4.3	Processes.....	59
4.4	Basic Instructions.....	60
4.5	Trap.....	62
4.6	The Memory Management Invariant (MMI).....	62
4.6.1	Verification of the Memory Management Invariant.....	64
4.7	Registers (0-5 and stack pointers).....	66
4.8	The PC and PS.....	68
4.9	Kernel Calls.....	69
4.10	Information Flows in Kernel Calls.....	70
4.11	Kernel Modification of Pages.....	70
4.12	Paging.....	73
4.13	Input-Output.....	76
4.13.1	DMA (Direct Memory Access) Devices.....	76
4.13.2	Buffered Devices.....	80
4.14	Pseudo Interrupts.....	81
4.15	Kernel Environment Maintenance.....	82
4.16	Proper Execution.....	84
4.17	Timing Dependent Confinement Channels.....	85
4.18	Security Policy Enforcement.....	86
4.19	System Correctness.....	86
 Chapter 5 - Encryption and Computer Network Security.....		88
5.1	Introduction.....	88
5.2	The Computer Network Environment and its Threats.....	89
5.3	Operational Assumptions.....	91
5.4	Encryption Algorithms and their Network Applications.....	92
5.4.1	Conventional Encryption.....	92
5.4.2	Public-Key Encryption.....	95
5.5	Error Detection and Duplicate or Missing Blocks.....	97
5.6	Block versus Stream Ciphers.....	98
5.7	Network Applications of Encryption.....	100
5.7.1	Authentication.....	100
5.7.2	Private Communication.....	101
5.7.3	Network Mail.....	101
5.7.4	Digital Signatures.....	101
5.8	Minimum Trusted Mechanism; Minimum Central Mechanism.....	102
5.9	Limitations of Encryption.....	103
5.9.1	Processing in Cleartext.....	104
5.9.2	Revocation.....	105
5.9.3	Protection Against Modification.....	105
5.9.4	Key Storage and Management.....	106
5.10	System Authentication.....	108
5.11	Key Management.....	111
5.11.1	Conventional Key Distribution.....	111
5.11.2	Centralized Key Control.....	113

5.11.3	Fully Distributed Key Control.....	115
5.11.4	Hierarchical Key Control.....	116
5.11.5	Public-Key Based Distribution Algorithms.....	119
5.11.6	Comparison of Public-Key and Conventional Key Distribution for Private Communication.....	124
5.12	Levels of Integration.....	126
5.13	Encryption Protocols.....	129
5.14	Confinement.....	132
Chapter 6 - Network Encryption Protocols.....		135
6.1	Network Encryption Protocol Case Study: Private Communication at Process-Process Level.....	135
6.2	Initial Connection.....	138
6.3	System Initialization Procedures.....	144
6.4	Symmetry.....	146
6.5	Network Mail.....	147
Chapter 7 - Digital Signatures.....		149
7.1	Introduction.....	149
7.2	Public-Key Based Methods.....	149
7.3	Reliable Digital Signatures.....	152
7.4	Network Registry Based Signatures - A Conventional-Key Approach.....	152
7.5	Notary Public and Archive Based Solutions.....	154
7.6	Comparison of Signature Algorithms.....	156
7.7	User Authentication.....	156
Chapter 8 - Conclusions.....		158
8.1	Computer Security.....	158
8.2	Conclusions and the State of the Art.....	160
8.3	Suggestions for Future Work.....	163
Bibliography.....		166

ACKNOWLEDGMENTS

I wish to gratefully acknowledge those who helped make this dissertation possible.

First, I wish to express my thanks to Professor Gerald J. Popek. His encouragement, guidance, friendship and criticism were essential, and I am sure this dissertation would not have been completed without his help.

Second, I wish to express my appreciation to the members of the Secure Systems and Software Architecture Group at UCLA, and in particular to those that contributed to the design and construction of the UCLA Data Secure Unix System.

Finally, I wish to thank my wife, Lanaii, and my friends for their love, support and encouragement.

This research was supported, in part, by the Advanced Research Projects Agency of the Department of Defense, under Contract Number MDA 903-77-C-0211.

VITA

August 22, 1948 Born, New York, New York

1970 B.S., University of California, Los Angeles

1971 M.S., University of California, Los Angeles

1966-1980 Various Positions, ARPA Research Project,
Department of Computer Science,
University of California, Los Angeles

PUBLICATIONS

Kline, C. S., A lisp interpreter in a paged environment, M. S. Thesis, Computer Science Department, University of California, Los Angeles, 1971.

Kampe, M., Kline, C. S., Popek, G. J., and Walton, E. J., The UCLA Data Secure Operating System Prototype, Computer Science Department, University of California, Los Angeles, Technical Report 77-3, July 1977.

Kline, C. S., and Popek, G. J., Encryption in computer network security, Computer Science Department, University of California, Los Angeles, Technical Report 77-2, April 1977.

Kline, C. S., and Popek, G. J., "Public key vs. conventional key encryption", Proceedings of the National Computer Conference, 48 (1979), AFIPS Press, Arlington, Va., 831-837.

Menasce, D. A., Rudisin, G. J., Popek, G. J., and Kline, C. S., A proposed architecture for the distributed secure system base, Computer Science Department, University of California, Los Angeles, Technical Report 79-10 (UCLA-ENG-7957), September 1979.

Popek, G. J., and Kline, C. S., "Verifiable secure operating system software," Proceedings of the National Computer Conference, 43 (1974), AFIPS Press, Arlington, Va., 145-151.

Popek, G. J., and Kline, C. S., "The design of a verified protection system," Proceedings of the IRIA International Workshop on Protection in Operating Systems, Rocquencourt, France, August, 1974, 183-196.

Popek, G. J., and Kline, C. S., "A verifiable protection system," Proceedings of the 1975 International Conference on Reliable Software, Los Angeles, Ca., April 21-23, 1975, 294-304.

Popek, G. J., and Kline, C. S., "The PDP-11 virtual machine architecture: a case study," Proceedings of the Fifth Symposium on Operating Systems Principles, Austin, Texas, October 1975.

Popek, G. J., and Kline, C. S., "Encryption protocols, public key algorithms and digital signatures in computer networks," Foundations of Secure Computing, R. DeMillo, et. al., eds., Academic Press, New York, 1978, 133-153.

Popek, G. J., and Kline, C. S., "Design issues for secure computer networks", Operating Systems, An Advanced Course, R. Bayer, R. M. Graham, G. Seegmuller, Eds., Springer-Verlag, New York, 1978

Popek, G. J., and Kline, C. S., "Issues in kernel design," Proceedings of the National Computer Conference, 47 (1978), AFIPS Press, Arlington, Va., 1079-1086.

Popek, G. J., Kampe, M., Kline, C. S., Stoughton, A. H., Urban, M. P., and Walton, E. J., UCLA Data Secure Unix - a secure operating system: software architecture, Technical Report 78-7 (UCLA-ENG-7854), Computer Science Department, University of California, Los Angeles, 1978.

Popek, G. J., Kampe, M., Kline, C. S., Stoughton, A., Urban, M., and Walton, E. J., "UCLA Secure Unix," Proceedings of the National Computer Conference, 48 (1979), AFIPS Press, Arlington, Va., 355-364.

ABSTRACT OF THE DISSERTATION

Data Security: Operating Systems and Computer Networks

by

Charles Steven Kline

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 1980

Professor Gerald J. Popek, Chairman

A great deal of attention has recently been focussed on the reliable protection of data in computer systems and networks. This dissertation examines the class of protection problems known as data security and discusses ways of reliably providing data security.

In the dissertation, the concept of data security is explained and a formal model of its meaning is presented. The UCLA Data Secure Unix Operating System is then described and the model is used to demonstrate, rather intuitively, that the system is indeed secure. The dissertation explains why such clear and intuitive demonstrations of security are a necessary complement to the more complex and formal approaches of others.

The dissertation also examines issues of providing data security in computer networks. Those networks add additional complexity to the data security problem which is

solved by the use of encryption. The dissertation explains how encryption can best be integrated into networks, and examines the tradeoffs in using conventional or public-key based encryption algorithms. The emerging issue of digital signatures is also discussed, and several possible implementations are presented. One rather surprising and important conclusion is that for most network security problems, conventional or public-key based encryption systems appear to be equivalent.

Chapter 1 - Computer security

1.1 Introduction

The Random House Dictionary defines security as "freedom from danger or risk; i.e. safety" [RAND80]. Achieving a degree of security requires a determination of possible risks, and the development of ways of dealing with those risks. Those ways fall into two basic categories: 1. preventing the risk from causing any damage or limiting such damage, and 2. making recovery from the damage cheap and efficient. In each case, however, there is usually some cost for the mechanisms which achieve security.

Computer systems are now used in many sensitive applications, and the list is constantly growing. Some examples include military command and control systems, electronic funds transfer and other banking and financial systems, automated offices, even personal computers. The use of computer systems presents a number of risks which can result in damage to individuals or organizations, although many of those risks are present even without the aid of computers. First, there are the physical risks such as those due to fire, earthquake or storm. These can result in loss or damage to equipment, and the resultant loss of the functions which the equipment provided. One cannot entirely eliminate these risks, but one can minimize the damages, for example by installing fire prevention and extinguishing

systems. In addition, one can arrange to have duplicate copies of the hardware, software, and data available for continued operation. Insurance is often available to help offset losses for this class of risks.

Another class of risks involves failures such as loss of power, hardware errors, or loss of communication facilities. One can attempt to protect against these risks with appropriate hardware such as uninterruptible power systems, fault tolerant hardware, or redundant facilities (e.g. multiple computers and communications lines).

A different kind of risk involves errors. For example, an error, whether human, software, or hardware, could result in the improper update of a credit record. The affected individual may sue or take other legal remedies to recover his damages. Protection against such errors involves error detection and correction procedures in the hardware, software and human environment (for example, using all keypunching verified). Additionally, insurance may sometimes be available to help offset the risks when these procedures fail.

1.2 System Security

This dissertation is concerned with a different class of risks than those mentioned above. The physical security and procedural problems briefly described above will not be

dealt with, nor the problems related to incorrect applications systems. Rather, it is assumed that the combination of hardware, system software and application software performs important functions, and it is these functions which must be protected. The protection of those functions will be referred to as System Security.

The functions can be of many different kinds. For example, the system may be dedicated to a single task, such as airline reservations, or it may be a general purpose time-shared computer system supporting many different users. The tasks being performed may involve sensitive data and the results may be quite important. Users desire the ability to rely upon the system to protect that data and to provide them with correct results. They may also desire to prevent a malicious user from interfering with their ability to get work accomplished, either by usurping system resources, or by directly affecting their computations. Users may even desire to be protected from their own carelessness or their own software errors. For example, many systems make it very easy to accidentally delete a file containing important information. If the file delete function is changed so that it merely marks files for later removal by the system, then users are given a chance to notice and correct their error.

The following sections briefly describe four common types of system security problems: 1. theft of service, 2.

denial of service, 3. data security, and 4. confinement.

1.2.1 Theft of Services

Even though the cost/performance ratio is continually decreasing, there is substantial motivation to obtain unauthorized and uncharged computer services in several possible ways. First, an unauthorized person may attempt to physically gain access to the areas where the system is located in order to steal equipment or service. Most of those unauthorized access attempts may be prevented through the use of guards, badges, locked doors, or other physical controls. However, in many environments it is useful to have terminals available in "open" (unprotected) locations, or to provide remote access capabilities via communication ports. These facilities make it difficult to prevent an unauthorized person from at least attempting to use the system. For this reason or when physical access controls fail, protection facilities are usually built into the system to prevent an unauthorized user from performing any functions. For example, it is common to employ a login scheme in which the user is required to identify himself by typing some form of name and password. Until a valid name and password are typed, the system will not perform any services for the user.

The password schemes in use today are a weak area in computer security. Many of the security breaches this

author has witnessed have been based upon poor choice and protection of passwords. These failures are partly due to the fact that the name and password are being used in an attempt to physically identify the person at the terminal. Clearly, an unauthorized individual possessing a valid user's name and password will be able to fool the system into assuming that the authorized user is at the terminal.

Once a user has been authenticated to the system, there are only a few ways by which he can attempt to steal service. First, he can attempt to exceed his authorized use of the system resources (cpu time, disk space, etc.). Such attempts can be blocked if the system contains secure mechanisms to allocate and control the use of system resources. Second, the user can attempt to prevent his usages from being properly recorded and charged. He can do this if the charging mechanism and information is not properly protected.

1.2.2 Denial of Service

Another risk which must be dealt with is called denial of service. If one user of the system can prevent a second user from getting his work accomplished, or accomplished in a reasonable time, then this second user has been denied service.

There are quite a number of denial of service issues.

First a user may use more than his share of system resources, thus delaying the work of some other user. Second, he may modify another user's files or interfere with the execution of programs, thus causing improper results. The first type described above will be called time denial of service and the second data denial of service.

Data denial of service is discussed below in the sections on data security and correctness. Briefly, the system should guarantee that a user's results only depend on his own actions and those of users authorized to interact with him.

Time denial of service is a complex issue. Is denying a user service for an hour followed by giving him all the service he needs a denial of service? Clearly the answer depends on the user's requirements. In some cases, the user could tolerate this delay. In other cases (for example a military missile control computer), the delay might be unacceptable.

It is clear that time denial of service is intimately involved with scheduling of the system resources (cpu, disks, etc.). It is difficult to design systems which guarantee specific real time response criteria, and this dissertation does not directly address such issues. However, many of the interesting time denial of service questions are present even in systems without such real time

criteria. In fact, can we even guarantee that each user receives some minimal service? The following is presented as a definition for such minimal service:

For a chosen time constant T and a unit of computation ϵ , during any period of time $>T$, each user must make at least ϵ progress towards his job's completion.

While showing that a system satisfies this definition does not guarantee any specific response time criteria, any system for which the definition can be shown to be true will have several important properties: the system can not crash or deadlock, nor can it deny service because of lack of resources. It is believed by the author that such systems can be built.

1.2.3 Data Security

Most of this dissertation is concerned with an important part of the computer security problems known as Data Security. Data Security deals with the problems of protecting information within a computer system. Intuitively, a system is data secure if it protects data in the system, controlling the accessing and movement of the information as it is processed in the system.

Conceptually, only data which has an effect on something in the real world (output to a terminal, orders to a missile control device, checks in a clearing house, etc.)

needs to be protected. Unfortunately, it is not possible to determine which data will eventually be output from the system, nor is it generally possible to check the validity of data as it is being output. Rather, the system must protect and keep track of the data from its entry into the system, through its storage and processing, until it is deleted from the system. Only in that way can users have confidence that no unauthorized individual has managed to process and output the data. In addition, this method allows the system to protect the data involved in a user's computation and guarantee that no other user has altered the results.

1.2.4 Confinement

Related to the concept of Data Security is the concept of Confinement. Confinement is the total control of movement of information, even involving subtle communication channels. Lampson [LAMP73] has identified three classes of communications channels. First there are the legitimate channels involving the normal program outputs of the various users. Next there are the storage channels in which the communication involves storage outside the user environment, such as through the use of system variables. Finally, there are the covert channels which include any communication methods not involving storage in the system. Examples of covert channels include program running time, power

consumption, electromagnetic radiation, etc. This dissertation describes methods of controlling legitimate and storage channels, but does not directly deal with covert channels.

1.2.5 System Correctness

The general concept of System Security described earlier includes the expectation by users of correct results. That expectation requires that the system, in addition to properly controlling access to data, correctly executes all functions provided. Constructing and certifying such systems is an extremely important but difficult problem. Work is in progress at several sites on this issue, but it is currently beyond the state of the art [POPE74c, NEUM77].

1.3 Implementation

Until recently, operating systems were designed without even data security considerations carefully examined. The resulting implementations contained numerous security flaws [POPE74c]. Many of the errors were in the so called "bug" category and could have been easily fixed by a careful system design audit. One common example occurred when a system implementer forgot to include a protection check of a parameter passed by a user to a privileged system routine. In that case, it is clear that adding the check would fix

the problem. Other flaws have not been so obvious to find or repair. Often it has been found that the fix to a system security flaw would require a major redesign of the system. For example, in the OS/360 system, crucial system data structures were stored in unprotected regions of memory. However, the simple fix of moving them into protected memory areas was not sufficient. Since the system design required certain user data also to be stored in protected areas, it was quite simple for a user to create data which could masquerade as the crucial system data. The proper solution required a more global system design change involving complex linking of the crucial structures with other system data so that the user could not "fool" the system with his masquerade structures.

As studies of existing systems have shown, it is clear that attempts to retrofit security into existing systems is generally infeasible [BISB75, CARL75, CARL76, HOLL76]. Rather, the security of the system must be carefully designed into the system from the beginning. Design principles to aid that process have been developing [POPE78c].

The numbers of data security errors in implementations of operating systems have been so large, and the problems so complex, that some have declared it impossible to build a truly secure system [COSC78]. The problem lies in showing

that there does not exist some action or sequence of actions which could result in security breech. Program verification methods appear to be the most promising approach towards solving this problem [POPE74c]. That is, with precise specifications of what data security is, one may be able to design and implement a system verifying formally that the system is secure with respect to the specifications.

The development of precise specifications for data security is presented in chapter 2. However, it is useful to provide some intuitive groundwork here.

Data security is the protection of information within the computer system. More specifically, it is desired to grant and deny various accesses (read, write, etc,) to pieces of information. However, the hardware does not provide mechanisms for this purpose, but rather primitive mechanisms which allow the protection of containers of information (e.g. memory pages, I/O devices). This leads to many problems.

First, the system must properly implement the higher level mechanisms which protect the containers. That is, the system must properly implement the abstract objects to be provided (for example, files), and protect those implementations from circumvention. Second, even with such correct implementations, the system must not have any errors which allow the user indirect access to the objects. For

example, if a user is not to be allowed access to a given file, the system must not have any errors which would allow him to gain access either directly or indirectly. In other words, in addition to preventing a user from directly accessing the file, the system must not leak the information.

For example, a well known flaw in the Tenex password mechanism allowed users to obtain other users' passwords, even though they were not able to directly access the password file [POPE75a]. The flaw resulted from the interaction of the system password check routine with the memory paging mechanism. By appropriately arranging a guess password across memory page boundaries, it was possible for a user to determine whether a password rejection occurred before or after a given group of characters. The user would then repeatedly guess the first character of the password until the rejection occurred after the first character. Thus, the first character was determined. Repeating the process would generate the rest of the characters. Thus, with a password of length n and with k possible different choices for each character of the password, the penetrator needed to test only k^n passwords, rather than the k^n tests assumed in password systems.

The above points out an important point in data security. If the system stores any information which is

based on a protected piece of information, then that stored information must also be protected. Part of the solution to this problem is to eliminate as much as is feasible conditions when the system depends on user's protected information. However, it also becomes clear that it is not just the control of access of information, but rather the control of movement of information which is important.

Finally, it is necessary to recognize that if there exists methods by which two users can communicate at all, then by accident or intent one may leak information to the other. In order to prevent this kind of leakage, one must attempt to control all communication channels. The problem of detecting and controlling all communication channels is the confinement problem, described earlier. In practice, this problem has been extremely difficult to solve in an efficient manner.

It should be mentioned here that the leaking of information from one user's protected data to another user except via covert channels is a data security problem. On the other hand, the unauthorized use of the system to communicate is a confinement problem. The point here is that if there exists a communication channel, it may be accidentally used by a user and information leaked. For a system to be data secure, all such legitimate and storage channels must be known, and it must be possible for a user

to write a program which can be guaranteed not to leak information (or at least be able to control the bandwidth of such leaks). In addition, those covert channels based on time, such as program running time, should at least have limited bandwidth.

1.4 Data Security Policies

This dissertation will attempt to distinguish between the mechanisms which detect and protect the movement of information, and the policies that control such movement. Different users have often requested different policies.

For example, one policy, known as the Military Security Policy[BELL73], assumes all data has a classification and all users have a clearance. Users are only to be allowed to read data with a classification less than or equal to their clearance. In addition, a user is not to be allowed to lower the classification of data without specific authorization. Another policy, the Access Control List Policy, assumes that there is a list associated with each group of data (known as an access control list) describing who may read or modify the data, and also describing who may read or modify the access control list.

While these are just two examples of Data Security Policies, they demonstrate some of the similarities and differences which are found among such policies. All Data

Security Policies answer the following two questions: 1. Is this access attempt allowed? and 2. Who can change the answer to question 1 and in what ways?

1.5 Certification

As the need for secure computer systems increases, some form of certification of the security of the system will become essential. Certification of the security of a computer system is a guarantee by a responsible party that the system is secure.

Part of the certification process is an examination of the system's design and implementation to discover possible weaknesses, and to discover possible damages which could occur via those weaknesses. Systems which claim to be data secure will require certification sufficient to guarantee that the probability of damage is acceptably low.

Initially, managers have had to use their own personnel to examine the security of their systems. Gradually, as expertise developed, firms began offering a security audit service. Eventually, it may be possible to buy computer security insurance. Insurance carriers would then be providing the certification.

This dissertation performs part of the certification process for the UCLA Data Secure Unix Operating System. Chapter 4 provides an intuitive and understandable

discussion of the ways that the Kernel of that system meets the data security specifications presented in chapter 2. That discussion is a necessary companion to the more formal verification methods employed by Kemmerer [KEMM79]. Those efforts resulted in complex specifications and tedious verifications which were difficult to comprehend. Here the concentration is on clearer, easier to understand presentations, while still formal enough to be translatable to the more tedious machine checkable versions.

While no system can be perfectly certified (since there is always some uncertainty at the atomic particle level on which the hardware circuits depend), it is necessary to carefully state the assumptions upon which the certification is based. In this dissertation, certification is based upon assumptions that the hardware works as specified and that the software systems have been compiled by correct compilers, assemblers, loaders, etc. Thus, the certification process becomes mostly one of examining the design and implementation of the algorithms which claim to enforce security properties and verifying that the properties are enforced.

1.6 Network Security

It has long been observed that as the cost per unit of equivalent computation became far less in small machines than large centralized ones, and as the technology of

interconnecting machines matured, computing would take on a more and more distributed appearance. This change is now happening. In many cases, users' data manipulation needs are best served by a separate machine dedicated to the single user, connected to a network of integrated data bases. Organizational needs, such as easy incremental growth and decentralized control of computing resources and information, are also well served in this manner. Multiprogramming of general application software in such an environment diminishes in importance.

As a result, the nature of the protection and security problem is beginning to change. Concern over the convenience and reliability of central operating system protection facilities is transferring to analogous concerns in networks.

The issues of protection in computer networks differ in several fundamental ways from those of centralized operating systems. One of the most important distinctions is the fact that the underlying hardware cannot in general be assumed secure. In particular, the communications lines that comprise the network are usually not under the physical control of the network user. Hence no assumptions can be made about the safety of the data being sent over the lines. Further, in current packet switched networks [KIMB75], the software in the switches themselves is typically quite

complex, often programmed in assembly language; one cannot say with certainty that messages are delivered only to their intended recipients.

The only general approach to sending and storing data over media which are not safe is to use some form of encryption. Suitable encryption algorithms are therefore a prerequisite to the development of secure networks, and considerable work has been developing in this area [LEMP79, NBS77, MERK79, RIVE77a, SIMM79]. However, equally important questions concern integration of encryption methods into the operating systems and applications software which are part of the network. The focus in this dissertation is on these latter issues, taking a pragmatic, engineering perspective toward the problems which must be settled in order to develop secure network functions. Cases where the safety of the entire network can be assumed are not discussed here, because issues in that environment are essentially those of distributed systems alone.

In networks, as in operating systems, there are several major classes of protection policies that one may wish to enforce. The most straightforward policy, satisfactory for most applications, concerns data security: assuring that no unauthorized modification or direct reference of data takes place. Highly reliable data security in networks today is feasible; suitable methods to attain this security will be

outlined in chapter 5.

A more demanding type of policy is the enforcement of confinement in the network. One commonly mentioned (and fairly easily solved) confinement problem is traffic analysis: the ability of an observer to determine the various flow patterns of message movement. However, evidence to be presented in chapter 5 indicates that the conditions under which confinement in general can be provided in a network are quite limited.

1.7 Dissertation Structure

The body of this dissertation presents an examination of some of the above issues in several environments. Chapter 2 presents a precise definition of data security, and presents a definition for one form of timing dependent confinement channels. Chapter 3 discusses the UCLA Data Secure Unix System prototype and chapter 4 examines how this prototype satisfies the data security definitions. Chapter 5 discusses the problems of data security and confinement in computer networks. Chapter 6 presents an example of an actual secure computer network design. Chapter 7 discusses the emerging issue of digital signatures. Finally, chapter 8 concludes the dissertation with an assessment of the current state of computer security, together with suggestions for future work.

Chapter 2 - Data Security

2.1 Introduction

Data security refers to the protection of data, i.e. information, as it is processed in a computer system. This chapter formally defines the concept of data security, and the related concepts of processes, information flow and confinement.

2.2 Processes

The discussion of data security begins with the notion of a process. A process is a potentially non-deterministic finite state machine.¹ A process can be represented by a set of states, an initial state, and a finite control. More formally,

$$P = (S, F, s_0)$$

where

S = finite set of states

s_0 = initial state

F = state transition function; $F: S \rightarrow 2^S$

It is useful to group parts of the state together into objects. An object o has two components: $o.value$ and $o.name$. $o.value$ is the value of the bits comprising object

¹. While processes are usually thought of as Turing machines, in reality all processes are finite, since the amount of storage on any real system is finite (although possibly very large).

o. $O.name$ is the name of object O . A process can then be defined by a set of objects together with the rules for state transition. Thus, a process state is represented by the values of the set of objects that make up the process. As the process executes, the values of those objects change according to the state transition function. This representation is more useful because it corresponds more naturally to the objects present in computer systems (e.g. files, memory pages).

2.3 Information Flow

As a process executes, it causes the transfer (or flow) of information among the objects that define the process. Intuitively, information flow occurs when the value of some object is changed based upon the values of other objects. This section defines information flow more formally, and the notions of read and write. The model to be described is an extension of that of Popek and Farber [POPE78b] and similar to that of Cohen [COHE78].

2.3.1 States and State Transitions

At any given instant in time, any process is in some state which is defined by the values of the objects which define the process. As the process executes, state transitions occur. State transitions for processes executing on real machines would occur from the actions of

the various devices of the system, such as the execution of instructions by the central processor or the movement of data by an input/output device. If s and s' are states, then define $s\{e\}s'$ to mean that the system changed from state s to state s' because of the occurrence of event e .

2.3.2 Read, Write, and Information Flow

In order to define information flow, it is useful to have a way of saying that two states are essentially the same, except perhaps for the value of an object or of a group of objects of interest. Thus, define a relation $E[N]$ on states, indicating a near equivalence.

For a list of names N and states s_1 and s_2 ,

$$\begin{aligned}s_1 E[N] s_2 \text{ if and only if } & (\forall o_1 \in s_1, o_2 \in s_2) \\ o_1.\text{name} = o_2.\text{name} \rightarrow & ((o_1 = o_2) \vee (o_1.\text{name} \in N))\end{aligned}$$

That is, $s_1 E[N] s_2$ if and only if states s_1 and s_2 are essentially equivalent, except possibly for the values of the objects with names in N .

With the above mechanism in hand, the following definition of information flow is presented. Assume a system is in state s_1 and state transition $s_1\{t\}s_1'$ occurs. Also assume that state s_1 has two components, a and b , which will be referred to as $s_1.a$ and $s_1.b$. Then, state transition $s_1\{t\}s_1'$ causes an information flow from a to b

(written $s_1[t]s_1':a \rightarrow b$) if
 $(\exists s_2, s_2' \text{ such that } s_2[t]s_2' \wedge s_1 E[(a)] s_2) \text{ such that}$
 $s_1'.b \neq s_2'.b$

Essentially, the above states that if two states s_1 and s_2 are essentially the same, with the possible exception of the object named a , and if the same state transition is applied to them, information flow from a to b occurs if b is different in the resulting states. Stated another way, two different values of the object a resulted in two different values of the object b .

The above definitions also apply to groups of objects. That is, it is trivial to extend the above for $s_1[t]s_1':A \rightarrow B$ where A and B are lists of objects.

The syntax $a \rightarrow b$ will be used to refer to information flow from a to b without specifying the state sequence causing the information flow, and $P:a \rightarrow b$ to refer to an information flow from a to b caused by process P .

When an information flow occurs from $a \rightarrow b$, it is said that object a is read and object b is written. It is important to notice that an object can be written in $a \rightarrow b$ even if its value does not change, since there must exist another value of object a for which object b would have changed.

The above definitions of information flow are

essentially the same as the "Strong Dependency" model of Cohen [COHE78]. Essentially, both models state that information flow occurs if variety (or change) in the values of input data results in variety in the values of output data.

Cohen has also demonstrated a second model of information flow. That model assumes a program schema and generates, deductively, the possible information flow relationships between the outputs of the program and its inputs. Cohen has shown the equivalence of these models. This latter model, and similar ones by Denning [DENN75], are useful when it is necessary to prove that a particular algorithm implemented in the system conveys information from only certain input variables to certain output variables. Such proofs are often part of the overall verification that a system meets security specifications and are discussed in chapter 4.

2.4 Domains

As mentioned earlier, data security is involved with controlling information flow. This dissertation will use the term domain to refer to a list of permissible information flows.² A process is said to be executed in a

². This definition of domain is slightly different than that used in other literature. In particular, other literature usually refers to a domain as the protection state in which a process executes, and includes the objects and operations which the process is allowed to

given domain. The intent is that only the information flows permitted by that domain should be allowed by the system to occur.

One representation for a domain is a list of object pairs, where information flow is only to be allowed from the first object to the second object of each pair. However, here a slightly weaker, but more convenient representation will be used in which a domain is represented as two lists: a read object list and a write object list. The interpretation of the two lists is that information flow is allowed from any object in the read list to any object in the write list. This representation is more convenient since it better models access policies provided in typical systems.

2.5 Data Security Policies

A data security policy provides the rules which control the way information flow relationships may be added or removed from domains. For example, the policy often referred to as the "Military Security Policy" requires that no information be readable by a process that does not have appropriate clearance, nor may highly classified information be transferred by a process to a less classified container without appropriate authorization. In this model, that policy could be enforced by requiring that no domain contain

perform.

an object on the read list with classification higher than the processes clearance level, nor that any object on the write list be classified lower than the highest classified object on the read list.

2.6 Systems

A System is the mechanism for executing processes. Each system has a set of objects and a set of rules for operations that may occur. It is the combination of these objects and rules which result in processes.

The rules describe the way that the next state transition for each process is selected and executed. For example, the instruction set of the central processor constrains many of the state transitions. The rules may exhibit non-determinacy, typically due to input/output. Thus, there is a set of computations (state sequences) which a given process may actually execute on a real machine, taking into consideration all the possible timing of non-determinacies. Although clearly finite, such a set may be very large.

For example, consider a process performing the following program:

```
integer a,b;  
a <- 0; b <- 0;  
start input/output;
```

```
{ assume that the input/output proceeds in parallel }
{ with the rest of the computation and eventually }
{ sets the value of variable a to 1 }
if a=1 then b <- 2 else b <- 3;
```

This example process has two possible computations which can occur, depending upon the exact timing of the I/O which sets variable a. In particular, representing a state as the pair of values of variables a and b, the potential set of computations is:

```
(0,0) -> (1,0) -> (1,2)
(0,0) -> (0,3) -> (1,3)
```

2.7 Proper Execution

A system is said to execute a process properly if, given the process' state transition function and the initial state of each object, the actual computation performed is one from the potential set of computations of that process. In other words, the computation which occurs is one which is allowed using the model of execution just presented above.

It is clear that a process which is not executed properly has had information input to it in a manner external from the process model just presented. In other words, if a process was not executed properly, then that process must have reached a state which could not occur by the process model of execution. Thus, some variable must have changed by some means other than by process execution

to a value which resulted in a new state sequence, and thus must contain new information. Generally, the source of this new information is not known. It may contain information the process should not have been allowed to access and result in information flows not allowed by the process' domain. Since it is impossible to analyze such external influences, all reasonable implementations of computer systems attempt to execute processes properly.

When a group of processes share objects, the rules for proper execution must be modified, since the group of processes, when executed together, may cause the computation state sequence for some process in the group to reach a state not in that process' original potential computation set. That is, when a shared variable is changed by one process, the states of all other processes which share that variable are changed, possibly to states which might not have been possible before. Under the definition of proper execution just presented, this problem would result in a "non-proper" execution for those processes sharing objects.

What becomes immediately clear from the above is that information flow is transitive. That is, when processes share objects, it is possible for information to flow into one of the shared objects via one process and then flow on to other objects via another process. This issue must be recognized when implementing security policies which allow

shared objects among processes.

Considering the above discussion, the definition of System Proper Execution can now be stated:

Given the system state at any instant in time, calculate the possible next states for the process which will next execute. The system properly executes if the actual next state for that process is found in the (just calculated) potential computation set.

2.8 Data Security

With the above definitions, it is now possible to define data security. A system is said to be data secure if:

1. As each process executes, only the information flows allowed by that process' execution domain occur,
2. The system is properly executed, and
3. The security policy is enforced.

2.9 Comparison of Models

The model presented above is similar to that of Popek and Farber [POPE78b] and Denning [DENN75]. All the models require that process information flows be restricted to those allowed by the security policy. The major difference between this model and the others is that the concept of

proper execution is introduced. In particular, the other models can be characterized as allowing the system to do anything to a process that the process would have been allowed to do to itself. Unfortunately, if the selection of the next operation is left completely unspecified, as it is in those models, then it is possible that the system could cause improper information flows.

For example, consider a process with a writable data area in its domain and operations which add a one bit or a zero bit to that area. If the execution sequence is left unspecified, it can not be stated that the system will not cause some arbitrary sequence to be written in the data area. In particular, the system could, through some unspecified means, cause protected information to be written in that area since each state transition of the process was allowed.

It can be argued that it would require a very strange system for such effects to occur. However, unless operation selection is specified, such behavior can not be ruled out. Proper execution specifies the selection of the next operation in a manner which conforms to the common ideas of instruction selection in typical computer systems.

2.10 Confinement

Information may flow from one process to another in

several ways. First, information flow may occur legitimately through the overlap of objects in the domains of the various processes. The model presented above prevents a data secure system from causing information flow via storage channels outside process domains.

A second method of information flow can occur via covert channels. While there remain covert channels external to the system (i.e. power consumption, radiation, etc.), within the system the only covert channel which remains is related to time. That is, by affecting the timing of the system, the choice of which computation is actually executed from the potential set of computations of each process may be changed. In particular, it may be possible for one process or a group of processes to affect that choice for another process or group of processes. It is possible that information can be passed using such mechanisms. Channels of this type will be referred to as timing dependent confinement channels. It is difficult to eliminate such channels, but many can be identified and their bandwidth controlled [POPE78c].

Chapter 3 - The UCLA Data Secure Unix Operating System

3.1 Introduction

The UCLA Data Secure Unix Operating System (UCLA DSU) represents the first successful attempt to build a data secure operating system based on the security Kernel concept [POPE78c]. Basically, the system is structured so that the security relevant modules are placed at the base level of the system, dependent only upon the interfaces provided by the bare hardware. In that way, their correct operation does not depend upon other code in the system. In addition, the security relevant software is carefully structured and minimized, in order to enhance the possibility that the security properties of the system may be formally verified as part of the security certification task. That goal required, as much as possible, architecting the system so that any software which was not security relevant was moved outside the Kernel. A complete discussion of security Kernels can be found in [POPE78c], and a comprehensive description of the UCLA Data Secure Unix System can be found in [POPE79]. A brief description is presented here in order to provide an understanding of the functions which belong in a security kernel and to provide a framework for the verification discussions of the next chapter.

3.2 System Structure

The system structure is presented in Figure 3-1. The system consists of a security Kernel, which implements an abstract machine, and processes which are run by that Kernel. The processes include all user processes and three special processes: the File/Policy Manager, the Scheduler and the Initiator/Dialoguer. The security of the system depends upon the security Kernel, the File/Policy Manager and the Initiator/Dialoguer.

Figure 3-1. UCLA Data Secure Unix Architecture

The system was designed and structured to allow a Unix compatible interface to be provided. In that way, most existing Unix software could be utilized without modification. That goal was largely successful and Unix software was tested on the prototype implementation.

The system was implemented for Digital Equipment Corporation PDP-11/45 and PDP-11/70 computers. The discussions below and in chapter 4 attempt to minimize the need for prior knowledge of the PDP-11 computer architecture since most of the system concepts are not machine dependent. Those readers wishing more detailed information can find it in the PDP-11 computer handbook [DEC75].

3.2.1 Security Kernel

The security Kernel is the heart of the Data Secure Unix System. It runs directly on the hardware, and turns the hardware objects (cpu, memory, I/O devices) into protected abstract objects (processes, pages, devices). The security Kernel has been carefully structured to be as small and simple as possible with the goal that its security properties may be formally verified.

The security Kernel provides a primitive capability system [FABR74]. Each process has a capability list maintained by the Kernel. The intent is that the actions of each process should be limited to those encoded in the capabilities. That is, the capability list is the internal representation of the process' execution domain. Thus, the major function of the security Kernel is to enforce the access restrictions represented in those capabilities.

3.2.2 File/Policy Manager (FPM)

The security Kernel provides protection on individual accesses according to the execution domain represented in the capabilities. However, it does not provide security policy enforcement. That function is left to a special process, the File/Policy Manager (FPM), which is the only process allowed to grant or revoke capabilities. Security policies can be implemented in that process by controlling the capabilities of user processes. For example, an FPM implementing the "military security policy" [BELL73] could refuse to grant a capability allowing read access to an object unless the user had a clearance sufficient to read the information contained in that object.

The Data Secure Unix File/Policy Manager also transforms the primitive page objects provided by the Kernel into files. In particular, user processes request access to files via requests to the FPM. The FPM performs any checks required to enforce the security policy and decides whether to grant capabilities allowing access to the pages of the file. That implementation choice was made primarily because files are the natural level for access controls. In addition, emulating the Unix functionality required controls at the file level.

The access control model implemented in the FPM in the UCLA DSU system is rather unique. Basically, files are

labelled with tags called colors which describe the data contained in the files. When a program attempts to move data from one file to another, the colors of the destination file must be a superset of the colors of the source file. The FPM allows two choices to satisfy that requirement: the attempt can be refused as an error, or the destination file colors can be forced to be the union of its colors and the source file colors. This mechanism allows a user to control and monitor the flow of information contained in his files. Thus, even using untrusted software which may have written sensitive data in improperly controlled locations, the user can prevent his data from being accessed by unauthorized users. A complete discussion of this implementation can be found in [URBA79].

3.2.3 Initiator/Dialoguer

The Initiator/Dialoguer is the last security relevant module. This module initially controls all terminals. When a user attempts to login, the Initiator/Dialoguer implements any user authentication protocols (such as password checks) and informs the FPM of the user's identification.

The Initiator/Dialoguer also provides a correct and secure channel for changing protection control information. The setting of the protection controls is crucial in any system. Errors in the setting of those controls will clearly result in security failures. However, most code

executed on the system will be uncertified. If protection changes are implemented utilizing uncertified code, and if there are errors in that code, then that could result in incorrect setting of the protection controls with disastrous consequences. For example, when one user attempts to grant another user access to a file, an error in the code used could result in the system granting the access to an unauthorized user. It is extremely important for users to be able to rely upon the protection controls which have been set. This problem is solved in UCLA DSU by prohibiting users from making protection changes via uncertified code. Rather, the user must switch his terminal to the secure Initiator/Dialoguer. That code is certified and allows a user to guarantee that protection changes are properly implemented [POPE79]. Since protection changes are relatively infrequent, it is believed that this minor user inconvenience is acceptable.

3.2.4 Scheduler

In order to minimize the security relevant software, as well as to allow considerable flexibility, the system is structured so that most resource scheduling is performed outside the Kernel. The Scheduler is the process which performs that resource scheduling.

The Scheduler process is executed whenever any other process releases control of the cpu. That occurs when the

process voluntarily releases the cpu (sleeps), when the process attempts to access a page not resident in main memory (page faults), or when the process has executed for a reasonable interval (time slice). When the scheduler process is run, it chooses the next process to be executed. Of course, the Kernel implements the actual process switching code since that code accesses critical hardware registers.

The scheduler also performs the paging function. The scheduler receives information from the Kernel about page usage and page faults, decides what pages to bring in and out, and issues the necessary system requests. The design thus allows considerable flexibility in implementation of scheduling and paging algorithms, functions which must usually be tuned to provide adequate performance.

3.2.5 User Processes

Other processes in the system are user processes. User processes access protected objects and cause processing and flow of information via the execution of PDP-11 machine instructions. Some of those instructions cause the invocation of Kernel implemented functions via hardware traps.

For convenience, user processes are usually divided into two parts -- one part implements the Unix compatible

interface, described earlier; the other part executes the user's code. On the PDP-11/45, this separation is made especially convenient and efficient since the machine has three hardware modes: kernel, supervisor and user. Thus the Kernel executes in the hardware kernel mode, the Unix Interface software in supervisor mode and the user code in user mode, simplifying addressing and interface issues.

3.3 Protected Objects

As described above, the Security Kernel implements an abstract machine upon which the processes are executed. That abstract machine contains protected abstract objects. The object list includes processes, pages, and devices.

3.3.1 Processes

Processes are an abstraction representing separately executable entities. A process consists of a capability list, a set of general registers, an area used for communication with the Kernel (known as an argument page), and a set of pending interrupts.

Processes access objects by their execution of PDP-11 instructions. The Kernel guarantees that actions which occur during those instruction executions are restricted to those allowed by the capabilities of the process. In part, the Kernel enforces this guarantee by utilizing the memory management hardware of the PDP-11 to perform access checks

on memory accesses. In addition, the Kernel provides routines which obtain control upon the various hardware trap and interrupt conditions. Those routines define additional functions which are part of the abstract machine specification. For example, one of the trap conditions provides an entry to the Kernel for the execution of several Kernel functions known as "Kernel calls". Those functions provide Kernel implemented code for certain security sensitive operations, for example, input/output.

3.3.2 Pages

Most storage in the system is in the form of pages. A page is a logical group of information which is individually protected by the Kernel. Pages are stored on random access storage devices (disks), but must be copied into main memory (swapped in) to be read or modified.

A user process can access a page only if it possesses a capability for the page with the appropriate access rights. In that case, the user process is allowed to cause the page to be associated with a given portion of its virtual address space by performing certain operations which will be referred to as the map function. Actual accesses to the pages occur when the process executes PDP-11 instructions that access that portion of its address space.

Pages move into main memory by having their contents

interface, described earlier; the other part executes the user's code. On the PDP-11/45, this separation is made especially convenient and efficient since the machine has three hardware modes: kernel, supervisor and user. Thus the Kernel executes in the hardware kernel mode, the Unix Interface software in supervisor mode and the user code in user mode, simplifying addressing and interface issues.

3.3 Protected Objects

As described above, the Security Kernel implements an abstract machine upon which the processes are executed. That abstract machine contains protected abstract objects. The object list includes processes, pages, and devices.

3.3.1 Processes

Processes are an abstraction representing separately executable entities. A process consists of a capability list, a set of general registers, an area used for communication with the Kernel (known as an argument page), and a set of pending interrupts.

Processes access objects by their execution of PDP-11 instructions. The Kernel guarantees that actions which occur during those instruction executions are restricted to those allowed by the capabilities of the process. In part, the Kernel enforces this guarantee by utilizing the memory management hardware of the PDP-11 to perform access checks

on memory accesses. In addition, the Kernel provides routines which obtain control upon the various hardware trap and interrupt conditions. Those routines define additional functions which are part of the abstract machine specification. For example, one of the trap conditions provides an entry to the Kernel for the execution of several Kernel functions known as "Kernel calls". Those functions provide Kernel implemented code for certain security sensitive operations, for example, input/output.

3.3.2 Pages

Most storage in the system is in the form of pages. A page is a logical group of information which is individually protected by the Kernel. Pages are stored on random access storage devices (disks), but must be copied into main memory (swapped in) to be read or modified.

A user process can access a page only if it possesses a capability for the page with the appropriate access rights. In that case, the user process is allowed to cause the page to be associated with a given portion of its virtual address space by performing certain operations which will be referred to as the map function. Actual accesses to the pages occur when the process executes PDP-11 instructions that access that portion of its address space.

Pages move into main memory by having their contents

copied from their location on disk (swapped in). When a page is in main memory, a user who is authorized to access the page will be allowed to access the area of physical memory in which the page resides. Pages move out of main memory when the physical memory is reused to contain some other page (when some other page is swapped into that area of physical memory). Before an area of main memory is reusable, the disk copy must be updated to reflect any changes which have been made to the page. Only when the disk copy and the main memory copy are identical (the incore page is "clean") can the memory be reused.

3.3.3 Devices

Input/output devices are divided into two categories: paged and assignable. Disks are usually divided into pages which are then accessed through the paging mechanism, just described above. Assignable devices are treated as protected objects for which a process may possess a capability. The Kernel provides input/output functions for accessing those devices.

Assignable devices, those which are treated as protected objects, actually are further subdivided into two classes: Buffered or Direct Memory Access (DMA). Buffered devices are those for which efficient usage requires the Kernel to provide buffering support. This class includes most terminals, since the hardware interface typically

allows only single character input and output. DMA devices (tapes, non-paged disks) are those which have hardware that can directly access memory. The Kernel allows those devices to input/output directly from user pages.

3.4 Kernel Specifications

This section presents a high level specification of the major Kernel data structures and functions. The verification discussions in the next chapter assume knowledge of the Kernel specifications presented here.

3.4.1 Kernel Data Structures

The Kernel has three data structures of interest: the Process Table, the Device Table, and the Incore Page Table.

3.4.1.1 Process Table

The Kernel maintains a process table for each process. The process table actually is in three parts, a table in Kernel space and two pages per process. The Kernel table contains the names of the two pages and other information which must be resident, for example pending interrupts.³ The other pages are the c-list page and the argument page. The c-list page contains the capability list of the process. The other page, known as the argument page, is shared by the

3. Actually, the pending interrupts are kept in a separate table to improve search efficiency. Those details are not relevant to the discussions at this level.

Kernel and the process and is used to pass arguments to Kernel functions and to receive return information. The Kernel also stores some process state information in that page such as general registers.

3.4.1.2 Device Table

The Kernel maintains a device table with an entry for each device. Basically, the device table is used to record information about input/output which is in progress, such as the process to whom interrupt information should be sent, or the page which is in use.

3.4.1.3 Incore Page Table

The Incore Page Table records the status of pages which are in main memory. A page can be "incore" or "swapping in", and can be "clean" or "dirty". A dirty page is one whose core image has been changed and whose disk image has not yet been updated to reflect the changes. In addition, the Incore Page Table contains a lock count for each page. The lock count indicates the number of I/O devices accessing the page.

3.4.2 Kernel Functions

The Kernel provides three basic functions. Each is listed with a short description.

1. System Initialization - The Kernel provides code to initialize the system. This involves initializing any devices, initializing Kernel tables, and causing the Scheduler and FPM processes to be marked runnable. The FPM has code in it to cause the initialization of the Initiator/Dialoguer and other user processes.
2. Trap Handling - The Kernel provides code to handle the various abort/trap conditions which can occur when a process is executing. Most such traps are merely recorded, and control is passed back to the supervisor portion of the process via the pseudo interrupt mechanism, described later. In that way, the Unix Interface may process the trap condition.

However, two traps are handled specially. The trap caused by the PDP-11 EMT instruction is used to request the Kernel to perform additional functions on the behalf of user processes. Those functions are known as Kernel calls. The Kernel implements those functions because they have security implications.

The other trap handled specially is the trap caused by the hardware memory management unit. That trap occurs when a process attempts to access beyond the limits of a page, or when the process attempts an invalid access. In most cases, that is an error and the process is informed as with other traps. However,

the system also uses that trap to perform the paging function. When the memory management hardware denies access to a page, it may be because the page is not in memory. An additional special case occurs when the page is resident in memory but the memory management hardware register has not yet been loaded to allow accesses. Thus, when the trap occurs, the Kernel examines the Incore Page Table to determine if the page is in core. If it is, the memory management register is updated and the process is restarted. That action is known as a register fault. However, if the page was not in memory a page fault has occurred. The process is suspended and the Scheduler is run.

3. Interrupt Handling - The Kernel contains the routines necessary to handle the various external interrupts which can occur. The only interrupts which can occur are those due to device I/O completions and those due to the clock. I/O completion interrupt routines perform any cleanup necessary (for example marking a page in memory), and cause a "pseudo interrupt" to be sent to the process performing the I/O. The clock interrupt routine increments counters to perform the time slice scheduling function and periodically causes the Scheduler process to be run. To simplify the Kernel design, code, and verification, the Kernel executes with Interrupts inhibited at all times.

Interrupts can only occur during the execution of processes.

3.4.3 Kernel Calls

The majority of the Kernel functions are those referred to above as Kernel calls. Kernel Calls provide functions which are necessary to implement a multi-user system and which are security relevant. The calls fall into several categories:

1. Process Scheduling functions: invoke, return
2. Input/Output functions: start-I/O, status
3. Paging functions: swap, reflect, zero-register
4. Inter-process Communication functions: send-interrupt,
set-interrupt
5. Capability Control functions: grant
6. Process Creation functions: init
7. System Debugging functions: log

Kernel calls are carefully structured to simplify the system design and verification. As a general rule, Kernel calls attempt to perform all checks before performing any actions. In that way, no "undoing" of actions must occur when various error conditions are encountered. As an

example, each call checks that all pages necessary are in memory before performing any actions. If any are not found, then a page fault is signalled. The page fault routine assumes that it can request the scheduler to obtain the needed page and then restart the call from the beginning without first performing any cleanup.

Kernel calls receive arguments in standard locations in the current process' argument page. That page is always in memory when a process is executing and simplifies the parameter passing mechanism. Return values are also returned in the argument page. Most calls return a true/false value indicating whether the call worked or failed. In addition, other return values are returned for many calls.

The following list describes the Kernel calls available, the actions the call performs, and the values returned. Each call is briefly described, followed in most cases by a more detailed description. To avoid repetition, it will be assumed that each call checks that appropriate pages are in memory and calls the page fault handler if they are not. In addition, only the return values other than the call worked/failed flag will be discussed.

Invoke The invoke call causes the system to switch processes. The invoke call is normally only issued by the Scheduler process.

Invoke saves the program counter (PC) and processor status (PS) of the running process in the Kernel process table, saves the general registers in the running process argument page, and loads the PC, PS, and general registers from the new process table and argument page. The memory management hardware registers for user and supervisor modes are set to special "null" values which deny any accesses and cause a trap on any reference.

Return

Return is used by processes to return from supervisor code (Unix interface) to user code. Return checks for pending interrupts to avoid race conditions if this function was not performed in uninterruptible code. Return also has an option which suspends the process and forces an invocation of the Scheduler process. When that option is used, the call is referred to as the sleep call.

Return first checks to see if the calling process has any pending interrupts. If it has some, then the call returns immediately having performed no actions. If the process has no pending interrupts, then

return checks to see if the "wants to sleep" argument was set. If so, the Scheduler is invoked. Otherwise, the PC and PS are loaded from the arguments provided after first modifying the new PS for safety (current mode, previous mode, register set, and processor priority).

Start-I/O

Start-I/O is used by a process to perform input output between a device and a page. Start-I/O checks capabilities for the device and page involved, verifies the parameters constrain the operation to the bounds of the device and page, performs any device dependent checks, and then starts the input/output. If the device is a DMA type, the appropriate start routine is called to physically load the device registers. In that case, the page which will be used is marked by incrementing its lock count. If the device is buffered, the Kernel copies data to (from) the buffer and then causes the device start routine to be called to empty (fill) the buffer.

Start-I/O returns various error codes or a value to indicate that the input/output is

in progress. In addition, most devices allow the user to receive a pseudo interrupt upon certain conditions (usually completion or error).

Status

The status call returns certain system status information. Status can be used by a process to find out the status of an input/output operation (busy, completed, errors), as well as by the Scheduler to find out the status of the cpu (pages which are dirty, processes which should be marked asleep or marked runnable) Status returns an error if the process does not possess an appropriate capability for the device on which status is requested.

Swap

Swap is used by the Scheduler to cause a page to be swapped in from disk, reusing an area of memory. The Kernel only allows the transfer to occur if the old page in that area of memory is clean, and if that page is not currently involved in an input/output operation (locked).

Swap checks that the page to be swapped in is not already in core or swapping in, that the page frame is of the appropriate

size, that the page occupying the page frame is clean, unlocked, and not swapping, that the page frame does not currently contain the argument page or clist page of the currently running process, that the device is not busy, and that the process has a capability allowing the swap operation. If any of these checks fail, an error is returned and the call terminates. Otherwise, an input/output is generated to cause the appropriate page to swap in, the page slot is marked "swapping in", and a code is returned indicating the input/output is in progress. If requested, an interrupt will be sent to the process when the input/output completes.

Reflect

Reflect is used by the Scheduler to cause a page in memory to be written back to disk. The Kernel only performs this operation if the page is actually dirty, and the page is not involved in an input/output operation (lock count is zero).

Reflect verifies that the process is allowed to perform the call, the page is resident in memory, that it is not locked or swapping, and the swap device is not busy.

If any of those checks fail, an error is returned. In the case where the checks succeed, but the page was already clean, the call returns with that information. Otherwise, the call starts an input/output operation to update the disk image, marks the page clean, and returns a code indicating the operation is in progress. The process may request to receive a pseudo interrupt upon the input/output completion.

Zero-register Zero-register is used as part of the page mapping function. Processes specify which pages they would like in their address space via a map table in the argument page. However, when a process changes the map table, it is also necessary to have the memory management hardware (relocation register) associated with that portion of the address space updated. Since the Kernel only looks at the map table upon register faults, the zero-register function forces the given relocation register to a value such that any access to that area would cause a register fault. The Kernel would then examine the map, find the new entry, and update the relocation register to point at the new page.

Send-Interrupt Send-Interrupt is used by one process to cause a notification of some event to one or more other processes. Effectively, each object in the system becomes a named entity on which interrupts can be sent and received. Each process specifies the objects for which it is interested in receiving interrupts by performing the set-interrupt function on capabilities for those objects. When a process performs a send-interrupt, it specifies a capability. All processes which have specified that they wish to hear about interrupts associated with the object specified in the capability receive an interrupt. The ability to send and receive interrupts on an object is controlled by access rights in the capabilities for the objects. Thus, the FPM has complete control over which processes may communicate using this mechanism.

In more detail, send-interrupt allows a process to name a capability, an access, and a byte of information. The access is checked to verify that it is a subset of the capability access, and that the capability allows sending interrupts. In that case, any

process which has an interrupt enabled on a capability for the same object, and with access enabled for a non-null intersection of the access the interrupt is sent with, will receive a pseudo-interrupt and receive the byte of information. It is possible that several interrupts on the same object will occur before a process happens to run to receive them. In that case, the interrupt information bytes are "or'ed" together. If the bits of the information byte have separate meanings, then this practice can be useful. For example, terminals send an interrupt using one bit for input available, and another for input overflow.

The interrupt implementation in the Kernel provides a separate table to store pending interrupts for each process. Thus, one process sending interrupts can not affect other processes, except those which are to receive the interrupts (no table overflows are possible).

Set-interrupt Set-interrupt allows a process to enable/disable pseudo interrupts on specific objects. The process specifies that

information by specifying which capabilities should have interrupts enabled or disabled. If enabling interrupts, the call verifies that the access requested is a subset of the capability access and that the capability allows receiving interrupts. In that case, the call also checks that the process has an unused Kernel interrupt table slot. If interrupts are being disabled, the call searches the interrupt table and frees any slot associated with that capability.

Grant

Grant is the function used by the FPM to grant or revoke a capability. The Kernel restricts the grant function to use by the FPM process only. Grant checks that the capability is reasonable, that the old capability in that slot is revokable (i.e. that no ongoing input-output operation is conditioned upon that capability), and then performs the revoke and the grant. A pure revoke is implemented by granting the NULL capability.

Grant has side effects on the process. It may force relocation register access to the special Null value (if some relocation

register was conditioned upon the old capability), and certainly will change the domain of the process receiving the capability. Grant also has the power to grant any access to any object in the system. Thus, grant may be executed only by certified code, the File/Policy Manager.

Init Init⁴ forces a process back to an initial, virgin state. Registers are zeroed, capabilities removed, etc. This call is used by the File/Policy Manager to initialize a process before reusing it. For example, each time the Unix "fork" primitive is executed, a free process is chosen, and an init performed. Since this function deals with user capabilities, it is restricted to the File/Policy Manager.

Log The log function was provided to simplify system debugging. Log prints a message on the operator's console terminal labelled with the identity of the process which issued the call. In a production version of the system, the log function would probably be replaced

4. Init is actually implemented as two functions, Init0 and Init1, to avoid certain internal problems. At this level that detail is irrelevant. The interested reader is referred to [POPE79].

by a more general operator communication mechanism. Log has no security implications. Any process may issue a log call.

3.4.4 Pseudo Interrupts

Processes are notified via a "pseudo interrupt" of certain events. Those events include I/O completions, messages from other processes, and certain trap/abort conditions. In each case, the process state (PC and PS) are saved in the argument page and the process is forced to enter the supervisor portion of the process (usually the Unix interface) at standard entry points.

The Kernel implements the pseudo interrupt mechanism by storing for each process a list of pseudo interrupts it is willing to handle (the interrupt table mentioned in send-interrupt above) and by searching that list when an interrupt must be sent. If a match is found, the Scheduler process is notified that the process should be run. In order to avoid certain race conditions and to simplify the Unix interface software, pseudo interrupts are automatically saved and only given to the process when it next attempts to execute in hardware user mode.

Chapter 4 - Intuitive Verification

4.1 Introduction

This chapter presents an intuitive verification method, and provides examples of its use in verifying the security properties enforced by the Data Secure Unix Kernel. In this chapter, the discussion assumes that the implemented algorithms correctly match the specifications presented in the previous chapter, and that the security policy is correctly implemented in the FPM. Thus, this part of the verification task largely involves a demonstration that each of the abstract objects is properly protected (process, pages, devices) and that each possible information flow is limited by the capabilities possessed by the processes.

This intuitive verification effort is meant to complement the work of Kemmerer [KEMM79]. That work demonstrated that the abstract type and mapping techniques of Alphard [WULF76] were suitable as a basis for formal verification of security properties. Thus, a high level model of security was presented along with suitable mapping functions enabling a mapping to the more concrete representations used in the code. Unfortunately, the reader is quickly engulfed in the tremendous amount of detail and complexity necessary by that formal model. The result is a rather unconvincing demonstration of the security of the system. While such methods are important, especially since

they may be eventually automated, it is still necessary to have understandable and convincing demonstrations of the security of the system. This chapter will attempt to perform part of that demonstration.

4.2 The Verification Methodology

The methodology used here is based on the Data Security definitions of chapter 2. Those definitions provide a process based model of Data Security. Thus the verification procedure here is to examine the information flows which processes can cause, and to demonstrate that they are restricted to those encoded by the process' execution domain.

The verification is presented in several parts. First, the functions which each process can perform are discussed with the intention of demonstrating that all basic information flows are controlled. That leads into a discussion of Kernel calls. Finally, with the control of information flow demonstrated, the discussion proceeds to Proper Execution, Security policy enforcement, and assumptions made along the way.

4.3 Processes

As was discussed in chapter 3, processes perform actions by executing PDP-11 instructions. Some of those instructions cause the movement of information directly from

data areas. Other instructions cause traps to the Kernel to perform the additional operations known as Kernel Calls. It must be shown that the information flows which occur as a result of each instruction execution are limited to information flows allowed by the process' execution domain, i.e. information must move only from objects for which the process has a read access capability to objects for which the process has a write access capability.

4.4 Basic Instructions

The PDP-11 instructions can be divided into two categories: the operate class and the trap class.

The operate class of instructions represents all the PDP-11 instructions which do not trap or abort.⁵ This class includes most instructions such as MOV, JMP, ADD, etc. On the PDP-11, the information movement which can be caused by the execution of these instructions is specified by the hardware [DEC75]. In particular, the PDP-11, like most computers today, contains memory management hardware to protect and relocate memory references. Each reference automatically selects an appropriate relocation register which both controls the accesses permitted as well as relocating the reference. On the PDP-11, there are separate

⁵. Actually, any PDP-11 instruction may trap or abort due to memory management violations. Such cases will be considered as two instructions: operate followed by trap, where the operate part reflects any functions which occur before the trap.

sets of registers for each of the system modes (kernel, supervisor, and user).

With the memory management hardware enabled, the objects the operate class instructions can reference or modify are as follows:

Reference:

1. Those areas of memory space addressable using the current mode memory management registers with read access.
2. Those areas of the memory space addressable using the previous mode memory management registers with read access.⁶
3. General registers (0-5) of the current set
4. Stack pointer (register 6) of the current mode
5. Stack pointer (register 6) of the previous mode⁷
6. PC (register 7)
7. Condition code bits of the PS

Modify:

6. MFPI, MFPD instructions access memory using previous mode.
7. MFPI and MFPD instructions can access previous mode stack pointer

1. The entire list above, with read changed to write
2. PS mode bits can be modified from Kernel mode to supervisor or user mode, or from supervisor mode to user mode.
3. The PS register set bit can be modified from set 0 to set 1
4. The PS trace trap bit can be written

4.5 Trap

The trap class of instructions includes those instructions which trap or abort. This includes the TRAP, EMT, IOT, and BPT instructions as well as any instructions which trap or abort because of errors (illegal operation, memory management abort, etc).

Most of the security implications of trap are discussed later in the section on pseudo interrupts. However, these instructions may also access some of the same objects as the operate class above before trapping. As an example, a MOV instruction may have incremented registers before failing on an invalid destination.

4.6 The Memory Management Invariant (MMI)

In order to show that the basic accesses made by an executing process are valid, it is necessary to show that

the process possesses a capability with appropriate access for the objects being read or modified. For accesses to memory pages, this is demonstrated by verifying the following invariant.

If the current mode bits (in the PS) are not Kernel mode, then

1. the previous mode bits (in the PS) must also not be Kernel mode, and

2. for each non Kernel memory management register, if the access field is not null, then

there must exist an entry in the Kernel incore_page_table such that

1. the process must possess a capability whose object matches the page in that entry,

2. the access field of the relocation registers must be a subset of the access of the capability, and

3. the relocation register must allow access only to a subset of the area of physical memory represented by that page.

Basically, this invariant requires that when a hardware relocation register allows access to an area of memory, then the process must possess a capability allowing access to the page which resides in that area of memory. The invariant requires that the previous mode not be hardware kernel mode to protect the kernel, since previous mode addressable memory is also available using the MFPD, MFPI, MTPD, MTPI instructions.

4.6.1 Verification of the Memory Management Invariant

The Memory Management Invariant can be demonstrated by induction in the following steps:

1. The system starts in Kernel mode
2. The MMI is true upon each exit from Kernel mode.
3. The MMI remains true while the system is not in Kernel mode.

Case 1: Trivially true by the hardware specifications.

Case 3: Only two types of system state transitions can occur while the system is not in Kernel mode. Those types are user instructions, and input/output device memory accesses. Neither of those two cases can change any of the conditions in the MMI as long as it is guaranteed that:

1. no non_Kernel memory management register ever allows access to the memory management register locations, the in core page table, the users capability list, or the device control registers (the upper 4k on the PDP-11)
2. no input/output is ever performed on those same objects.

These two conditions are discussed under the section on Kernel environment maintenance.

Case 2: This case requires examining the state upon each exit from the Kernel. If the invariant was true upon entry to the Kernel, it would remain true upon exit from the Kernel unless the Kernel changed some condition of the invariant. An examination of the Kernel routines shows that only the following Kernel routines modify those conditions. Each is described below with its relationship to the MMI.

Grant This routine revokes a capability. Since that capability may have been supporting some relocation register, revoking that capability could invalidate the invariant. Therefore, the revoke capability routine checks to see if any relocation register is conditioned upon that capability and, if it finds one, sets the relocation register to null access.

Swap This routine changes the incore page table. For the same reasons as grant above, swap checks to see if any relocation register is conditioned upon that incore page table entry and, if so, sets the access to null.

Invoke This routine changes the process parameter of the invariant. That action almost certainly would invalidate the invariant. Thus, the invoke routine automatically sets all the non Kernel relocation register accesses to the special null

value.

Register-fault This routine handles a memory management fault. As described in chapter 3 in the section on the map function, when this routine finds that the fault is for a page which is in memory, and for which the process has the appropriate access in its capability, the routine constructs a relocation register to point at that page in memory. This routine is the only case which sets a relocation register access to non null. The routine very carefully constructs the relocation register to satisfy the invariant. It is straightforward code verification to show that the invariant is satisfied. However, that verification depends upon the correctness of the incore page table and the process' capability lists. That is discussed in a later section.

4.7 Registers (0-5 and stack pointers)

Since the hardware allows each process to access the registers, it must be shown that no data security violations occur via these hardware objects. This requirement is implemented by providing each process with its own logical set of registers. It must then be shown that these logical registers are properly implemented. That is, each process finds in the registers the last values written by that

process.

The implementation of the logical set of registers is straightforward. At any given moment, only one process can actually be executing on the cpu. That process' logical registers are loaded in the real machine registers. When the cpu is switched to another process, the register values are saved in the process table, and the new process' registers are then loaded. (see invoke Kernel call in chapter 3)

In more detail:

1. At system initialization, all process registers (in the process tables as well as the real registers) are set to default initial values (constants). Thus, they contain no user information. In addition, the system is in Kernel mode.
2. The only time the Kernel changes the hardware user registers is in the invoke routine.
3. Invoke merely copies the current registers into the current process table and then loads the registers from the new process table.
4. The only way the process table register values change is in the invoke routine, and in the init routine.

5. The init routine sets the process table registers to default values again, thus conveying no information.⁸
6. The only other way the real registers can be read or written is when some process is running. In that case, by induction, it is clear that they contain the last values written by that process and will continue to only contain information as defined by the PDP-11 instruction set.

Thus, upon exit from the Kernel the first time, the process registers contain constants. From that time forward, the process registers contain the last values written by the process.

4.8 The PC and PS

These objects are treated the same as other registers with the exception of pseudo interrupts. That is, each process has a logical PC and PS which are saved across process switches. Pseudo interrupts occur upon some supervisor or user traps, and upon certain events such as I/O completions. The security implications of changing the PC and PS bits by the pseudo interrupt mechanism is discussed in the section on pseudo interrupts.

⁸. other than time. (see discussion of timing channels)

4.9 Kernel Calls

The trap instructions cause entries into the Kernel at particular entry points defined by the trap hardware of the PDP-11. With the exception of the Kernel call (EMT instruction) and certain memory management traps, the traps are merely reflected by the Kernel back to the supervisor. This action is discussed in more detail in the section on pseudo interrupts.

The Kernel call causes the Kernel to perform various actions on behalf of the process. These actions include changing processes, performing I/O, and other functions. In general, these functions are implemented in the Kernel because they either involve use of the hardware registers which, if not controlled, would allow unlimited access to memory or devices, or because they involve actions on Kernel supported objects such as capability lists.

The Kernel, upon entry, examines the trap to determine which call is requested and calls the appropriate routines. Upon return from those routines, the Kernel returns to the current process (which may not necessarily be the same as the one upon entry to the call, for example, after a sleep or invoke call).

The Kernel calls are largely structured in a particular way which is described here. Basically, each call performs

certain validity checks upon its parameters. For example, start-I/O verifies that the capability specified contains a device capability. In order to simplify the code, a general rule followed is that all checks are performed before any actions. In that way, no "undoing" of side effects must be considered upon error conditions. For example, several calls check that certain pages are in memory. If the pages are not in memory, then the call signals a page fault. In that case, the call is aborted; since no side effects have occurred, the call may be restarted in its entirety when the page is in memory.

4.10 Information Flows in Kernel Calls

Since the Kernel has complete control over the system, and thus global access to data, it must be demonstrated that the Kernel itself does not cause improper information flows. Since the Kernel does not change user registers, with the exception of pseudo interrupts (discussed later), only through page modification can the Kernel pass information.

4.11 Kernel Modification of Pages

Pages are never modified by the Kernel except in a few special cases:

1. Pages are copied in and out of memory (swapped),
2. Certain pages which no user can read are updated by the

Kernel (process table pages),

3. The pages known as argument blocks are updated by the Kernel.

The demonstration that paging is correctly implemented is discussed in a later section. The discussion which guarantees that processes can not access the Kernel pages (process table pages) is discussed in the section on Kernel environment maintenance.

Thus, the only case where a Kernel modification of pages can affect a process is the use of Kernel argument block pages. However, this case is somewhat complex. Since each Kernel call, as well as pseudo interrupts, return information in the argument blocks, it must be guaranteed that no unauthorized information flows occur via those mechanisms. Basically, this involves showing that the information is only a function of other information the process had a capability to read. In other words, no new information flow is added that the process could not have already accessed directly. The verification of this last task must be performed for each individual Kernel function which modifies the argument block.

Many of the Kernel calls are memoryless. That is, their effects depend only upon the parameters passed to the call. The values returned, as well as the side effects, are independent of other calls which have been performed. For

example, zero-register sets a given relocation register to null access. Its effect is independent of prior calls by this or other processes and it returns no information.

However, there are calls which do return information from the Kernel, and therefore may be passing information. The status call returns status of an I/O device. However, only processes authorized to access the device can obtain information using the status call. Thus, this channel is controllable by the FPM.

Certain calls can return status of the entire system. Those calls, however, are all controllable by capabilities, and are expected to be restricted to the Scheduler process by the FPM. In particular, the status call returns status of the cpu (i.e. pages in and out of memory, status of processes, etc.), the paging calls (swap and reflect) return information about the status of the cpu (pages in memory), and the invoke call returns information about the status of the cpu (pages in memory). All those calls would normally be restricted to the Scheduler.

The grant call is a very special call. It has the power to grant any capability to any process. Thus, its use must be carefully controlled. The Kernel restricts that call to the FPM process.

The only other calls which may return information not a

function of the parameters of the call are the pseudo interrupt calls (send-interrupt and set-interrupt). The security implications of those calls are discussed in the section on pseudo interrupts.

4.12 Paging

Paging was briefly discussed in chapter 3. Each page frame of memory may contain a page. Pages are brought into page frames via the swap function. Pages are used either by mapping them into some portion of a process' address space, or by using them via a start-I/O call.

Page frames can only be reused when their contents are clean, i.e. the disk copy matches the core copy. Pages which are dirty are reflected back to disk via the reflect operation.

The correctness of the routines in the Kernel which perform the swap and reflect functions is crucial, not only because they interact with other Kernel routines (via the in-core page table), but also because certain Kernel data structures (the capability lists) are kept in pages. That Kernel data must satisfy certain properties for other parts of the security proof.

The following discussion demonstrates that data values in pages are kept consistent over time by the Kernel, and that the paging functions are properly implemented.

1. A given area of memory contains only zero or one pages.
2. The same page is never in two places of main memory simultaneously.
3. If a page is reflected from main memory to disk at time x, removed from memory, and then next swapped back to some area of main memory at time y, then the contents of the page are identical at times x and y (pages are properly written and read from disk).
4. Pages only change values in main memory when explicitly accessed by instructions or I/O devices.

These four assertions form the basis for the verification of the swap and reflect Kernel calls, the page table updating routines, and the memory management register updating routine. That verification proceeds roughly as follows:

1. A page is considered "incore" if the incore page table has an entry for the page and the state is incore. Initially the entire incore page table entry is marked so that no pages are incore.
2. The incore page table has an entry for each area of physical memory. Thus, to show that an area of memory contains only zero or one pages is simply to show that only one page, at most, is placed in that area. That

property is shown by showing that when a page is swapped into memory, the incore page table is updated, and that the swap routine will not swap a page into an area without first removing the page that was there.

3. Showing that a given page is never in two places in main memory simultaneously is performed by showing that the only routine which places a page in main memory is the swap routine, and that that routine will not place a page in main memory if it is already in main memory.
4. Showing that when a page is written to disk and then read back into core results in the same values depends on showing that the disk copy did not change, and that the input/output routines properly write/read the disk.
5. Showing that pages do not change values while on disk requires showing that no input/output will occur to the location on disk which contains a page. That property results from the page to disk mapping routines which are designed so that each page has a unique area on disk, and that the input/output routines properly read/write to that area of disk.
6. Finally, showing that pages in main memory only change through explicit access requires examining the system to find those places in the system where it is possible for pages to change values. That has already been done

in the previous sections, with the exception of device input/output, which is discussed next.

4.13 Input-Output

Input-Output calls are used to perform transfers from process memory to various I/O devices. The I/O devices fall into two classes: DMA and buffered. DMA (direct memory access) devices perform I/O directly from process memory pages. Buffered devices (typically terminals) are buffered by Kernel software.

4.13.1 DMA (Direct Memory Access) Devices

DMA I/O devices have the property that they can (potentially) access any locations in real memory. Fortunately, I/O devices also have the property that the set of accesses they will make is very well defined.

DMA devices on the PDP-11 have the following properties. When idle (not making accesses), they remain idle until specifically started. Once started, via loading special device registers, they will make a finite set of memory accesses which can be determined in advance based on the values loaded in the device registers. During this time, the device may cause one or more interrupt requests to the cpu. After these finite accesses, the device will return to the idle state.

The above properties make implementation and verification of I/O possible. Basically, it is demonstrated that the information flows the device will cause are valid at the time the device is started, and that they will remain valid until the device returns to the idle state.

The following steps are necessary to demonstrate the security of DMA I/O:

1. Show that devices which are idle remain idle unless specifically requested via a start-I/O from a user process
2. Show that at device start time, the process requesting the I/O had a capability for the device and a capability for the page which the device will access such that the information flow is valid.
3. Show that these capabilities remain valid until the device is idle.
4. Show that the device will remain within the bounds of these capabilities.

The demonstration of the above is now described.

1. As mentioned earlier, the Kernel environment maintenance assertions guarantee that the I/O device registers will not be accessed except in those Kernel routines which explicitly utilize them. An examination

of the Kernel shows that the only routines which load values in device registers which could actually cause I/O to occur are the various cases of the start-I/O Kernel call.

2. To demonstrate that the process possesses appropriate capabilities at the time that the I/O is started requires a detailed examination of the start-I/O routine. First, it must be demonstrated that the values loaded for a given device will only cause the I/O device to access a given area of memory. That part of the verification task is device dependent. Next it must be shown that this area of memory corresponds to a single abstract object, a page. This requires a code verification showing how the start-I/O code bases the values loaded on the incore page table entry. Finally, an examination of start-I/O shows that the user contains a capability granting appropriate access to this abstract object. This examination of start-I/O also demonstrates that the I/O will remain within the bounds of the capabilities.
3. To show that these accesses will remain valid during the I/O is simply to show that the necessary properties cannot change during the I/O. In particular, the only ways in which this I/O could become invalid would be if the capability upon which it is based, or the abstract

object which corresponds to the real memory to be accessed change. The only routines which change these are the grant (revoke) capability routine, and the swap routine. Both those routines contain checks which guarantee that they will refuse to perform an action which would invalidate running I/O. In particular, the grant capability routine checks to make sure the capability to be removed is not supporting some I/O operation. This action is implemented by a combination of the start-I/O routine storing the process number and capability slot number for the capability supporting the I/O and by the grant capability routine checking that stored information. The swap call also checks to make sure it is not modifying the physical to logical mapping for an area of physical memory which is being used by an I/O device. This is implemented by the start-I/O routine incrementing a lock count in the incore page table for the page (area of real memory) being used and the I/O completion routine (an interrupt routine typically) decrementing this count when the device is idle. Finally, the swap routine will not change the mapping (incore page table) for any page whose lock count is non zero.

Thus, the above guarantees that the information flows were allowed at start time and remained valid while the I/O device was making accesses.

4.13.2 Buffered Devices

These devices are buffered in special areas of the Kernel. On input, information is transferred from the device to the buffer and then to the user. On output, information is transferred from the user to the buffer and then to the device.

As with DMA devices, the user should only be allowed to read or write a device if he has the appropriate capability for that device. In addition, he must possess a right for the information being output, or for the place to store the information being input. It must be shown that the Kernel only transfers the information under those conditions.

The actual verification of this proceeds roughly as follows:

1. For any buffer, the device to which the contents are to be written, or from which the contents were read is known. (The Kernel must therefore very carefully keep track of this information. In the UCLA Kernel, it is statically associated at compile time.) In particular, the Kernel must not write data to a buffered device except from an appropriate buffer.
2. When a process requests input/output to a buffered device, the Kernel checks the device capability to be sure the user is allowed to use the device in the

requested manner and the data capability, offset, and count to make sure the data area is legitimate. If both of these checks (and any other device dependent checks) succeed, then the data is copied to or from the buffer and the user area. (These are the same parameters and checks used for DMA I/O, and, in fact, the same code is used.)

3. Thus, user data is moved to or from a buffer appropriately labelled for the device and data is moved to or from the device only from such a buffer.

4.14 Pseudo Interrupts

Pseudo Interrupts occur both synchronously and asynchronously. Synchronous pseudo interrupts occur when a process traps. The Kernel catches the trap, stores in the argument page the PC and PS at the time of the trap and a code indicating which trap occurred, and enters the supervisor portion of the process. Asynchronous pseudo interrupts are sent when I/O completions occur and when processes interrupt other processes via the send-interrupt call.

Synchronous pseudo interrupts can be considered as additional machine instructions. Their effect is well defined, and they clearly pass no information that the process could not have accessed in other ways.

Asynchronous pseudo interrupts, however, have two side effects which have potential security implications. First, when an input/output operation is requested, the process computation becomes a parallel computation, as described in chapter 2. The parallel computation, when sequentialized, results in a set of possible computation state sequences depending on the exact timing of the parallel computations. The pseudo interrupt is an extension of that case. That is, the data that the pseudo interrupt will return is fixed; the variable is the time at which the pseudo interrupt occurs. Thus, this case of asynchronous pseudo interrupts conveys no information other than time, and is a case of a timing dependent confinement channel, discussed later.

The other case of asynchronous pseudo interrupts is when one process sends an interrupt to another using the send-interrupt call. Clearly, information can be passed using this mechanism. However, the ability to send and receive interrupts is checked by the Kernel. Thus, this channel is controllable by the FPM, in the same manner as shared files. Hence, no new information flow is created by this mechanism.

4.15 Kernel Environment Maintenance

The verification discussions above assume that the Kernel code and data is uncorruptible. That requirement must itself be provided by the Kernel. That is, the Kernel

must maintain its own environment. This requirement is demonstrated by showing that the Kernel code is "well behaved" and that the code and data can not be affected.

To guarantee that the Kernel code is well behaved requires several properties to be true. First, the Kernel must be compiled, linked, loaded, etc. via correct compilers, loaders, etc. Second, the Kernel stack must be sufficient to contain the maximum nesting which can occur in the Kernel. Third, the Kernel must guarantee that all array indexing is within bounds.⁹ The UCLA Kernel is not interruptible and uses no pointer variables so that these two issues present no problems.

To avoid corruption of the Kernel simply requires that no I/O is performed into Kernel code or data space, and that no user is ever able to directly access the Kernel code or data area. That property is maintained by the code which loads user relocation registers, and by the code involved in I/O. In particular, the incore page table does not contain entries for the pages of memory representing Kernel code, data, or the device control registers. Thus, a user process cannot name such a page and thus cannot access those pages.

^{9.} The UCLA Kernel is written in a version of PASCAL which does not include automatic array checking either at compile or run time.

4.16 Proper Execution

The certification of proper execution is performed in two steps: 1. Definition of what a proper execution for a process is, and 2. Showing that the system actually properly executes the process.

The definition of proper execution of processes basically involves the definition of instruction selection and execution. As discussed earlier, the system defines a process to be the set of objects connected to it and several state variables defining the operations in progress. The next instruction is selected non-deterministically either to be the movement of information from (to) memory to (from) an I/O device, or the execution of the next Kernel extended PDP-11 instruction. As has been discussed in previous sections, the PDP-11 instructions have well defined effects. The Kernel supported instructions also have well defined effects. The I/O activity, as constrained by the kernel, is also well defined in its behavior. Thus, the proper behavior of the process, while non-deterministic, is quite well defined.

Thus, to show proper execution, one merely must show that the state of the process only changes due to one of the above conditions. That has already been shown in earlier sections:

1. The state of the CPU is maintained - general registers, PC, PS, SP.
2. The state of the I/O is maintained
3. The state of the objects is maintained (objects only change due to user I/O, user instructions)

4.17 Timing Dependent Confinement Channels

There are several timing dependent confinement issues. First, since the timing of events is not well defined, potential confinement channels exist. For example, with most reasonable schedulers, one user can certainly effect the speed at which other users computations are performed. The users can sense this behavior through the various clocks provided as well as the fact that certain operations will change in time line. For example, even if there were no clock operations available from the Kernel, users could sense changes in the points at which pseudo interrupts occurred. Even if those operations were made deterministic, a user process connected to a terminal can get time from the user as input.

Most of the remaining timing dependent confinement channels involve resource allocation. Since all systems have finite resources, a user will be able to sense when a resource has run out. In the UCLA DSU system, all such channels are controllable by the FPM since it allocates all

resources. Thus, it is possible to implement various methods to either block or put noise in those channels.

4.18 Security Policy Enforcement

This chapter has largely ignored security policy enforcement. Rather, the chapter has attempted to demonstrate that all the information flows (with the exceptions of certain low bandwidth timing dependent confinement channels) are controlled.

With the control of information flow, it becomes a matter for the policy manager to control the granting of capabilities in order to maintain a security policy. A complete discussion of this issue is beyond the scope of this dissertation, and the interested reader is referred to [POPE74a, URBA79].

4.19 System Correctness

Implementing a correct security policy manager, as well as secure applications code, requires more from the system then just the control of information flow. Rather, these "higher" levels of software will have additional security properties which must be demonstrated. Unfortunately, to demonstrate the security of a given layer requires a correctly implemented abstract machine on which that layer is constructed.

For example, since the policy manager is constructed on top of the Kernel, the Kernel functions must be "correct" in order to provide a basis for verifications of the policy manager. Similarly, a verification of a property at level n will usually require the prior verification of the correctness of all layers 1 to $n-1$. Thus, a complete security verification of the system would require a specification of the abstract machine assumed by the policy manager, and a verification that the Kernel implements that abstract machine correctly. Then, the policy manager security properties could be verified. Finally, any secure user layers would require a similar specification of the environment presented by both the Kernel and the policy manager, and a verification of the correctness of the implementation.

As this chapter and the work by Kemmerer [KEMM79] and others have demonstrated, the ability to efficiently specify and verify properties of large systems is still an extremely difficult problem. Yet, such an ability is becoming more and more important, especially in the area of computer security. It is hoped that research will continue into specification and verification languages and techniques so that practical systems can be constructed and completely verified.

Chapter 5 - Encryption and Computer Network Security

5.1 Introduction

The dramatic decreases which have occurred in the cost/performance ratios for computer equipment are resulting in significant changes in the ways computers are used. It has become quite feasible to dedicate a machine to only a few functions, interconnecting it with other machines performing other functions. As a result, the data security problem has grown beyond the bounds of the central operating system and into the computer communications network.

While there are still central operating system security problems which remain in such systems, the new environment presents new security threats which require a different approach for solution. In particular, the communications media can not be assumed to be under the control of the network user.

The following sections describe design problems and alternatives available for the design of secure networks and discuss their utility with respect to data security and confinement. In addition, chapter 6 presents an illustrative case study of the use of these methods in an actual computer network.

5.2 The Computer Network Environment and its Threats

A network may be composed of a wide variety of nodes interconnected by transmission media. Some of the nodes may be large central computers; others may be personal computers or even simple terminals. The network may contain some computers dedicated to switching message traffic from one transmission line to another, or those functions may be integrated into the general purpose machines which support user computing. One of the important functions of computer networks is to supply users with convenient private communication channels similar to those provided by common carriers. The underlying transmission media, of course, may be point to point or broadcast. Considerable software is typically present to implement the exchange of messages among nodes. The rules or protocols governing these message exchanges form the interface specifications between network components. These protocols can significantly affect network security concerns, as will be seen later. In any event, because of the inability to make assumptions about the communications links and switching nodes, one typically must expect malicious activity of several sorts.

1. Tapping of Lines. While the relevant methods are beyond the scope of this discussion, it should be recognized that it is frequently a simple matter to record the message traffic passing through a given

communications line without detection by the participants in the communication [WEST70]. This problem is present whether the line is private, leased from a common carrier, or part of a broadcast satellite channel.

2. Introduction of Spurious Messages. It is often possible to introduce invalid messages with valid addresses into an operating network, and to do so in such a way that the injected messages pass all relevant consistency checks and are delivered as if the messages were genuine.
3. Retransmission of Previously Transmitted Valid Messages. Given that it is possible both to record and introduce messages into a network, it is therefore possible to retransmit a copy of a previously transmitted message.
4. Disruption. It is possible that delivery of selected messages may be prevented: portions of messages may be altered, or complete blockage of communications paths may occur.

Each of the preceding threats can, in the absence of suitable safeguards, cause considerable damage to an operating network, to the extent of making it useless for communication. Tapping of lines leads to loss of privacy of

the communicated information. Introduction of false messages makes reception of any message suspect. Even retransmission of an earlier message can cause considerable difficulty in some circumstances. Suppose the message is part of the sequence by which two parties communicate their identity to one another. Then it may be possible for some node to falsely identify itself in cases where the valid originator of the message was temporarily out of service.

More and more applications of computer networks are becoming sensitive to malicious actions. Increased motivation to disturb proper operation can be expected: consider the attention that will be directed at such uses as military command and control systems (by which missile firing orders are sent), or commercial electronic funds transfer systems (with daily transactions worth hundreds of billions of U.S. dollars).

5.3 Operational Assumptions

The following discussions of protection and security in computer networks are based on several underlying assumptions:

1. Malicious attacks, including tapping, artificial message injection, and disruption are expected.
2. The insecure network provides the only available high-bandwidth transmission paths between those sites which

wish to communicate in a secure manner.¹⁰

3. Reliable private communication is desired.
4. A large number of separately protected logical channels are needed, even though they may be multiplexed on a much smaller number of physical channels.
5. High-speed inexpensive hardware encryption units are available.

It is believed that these assumptions correctly mirror many current and future environments. The next sections outline properties of encryption relevant to network use. Those interested in a deeper examination should see [LEMP79, SIMM79]. After this brief outline, the discussion of network security commences in earnest.

5.4 Encryption Algorithms and their Network Applications

5.4.1 Conventional Encryption

Encryption provides a method of storing data in a form which is unintelligible without the "key" used in the encryption. Basically, conventional encryption can be thought of as a mathematical function

10. It will turn out that some presumed secure and correct channel will be needed to get the secure data channel going, although the preexisting secure channel can be awkward to use, with high delay and low bandwidth. Distribution of the priming information via armored truck might suffice, for example.

$$E = \underline{F}(D, \underline{K}),$$

where D is data to be encoded, K is a key variable, and E is the resulting enciphered text. For F to be a useful function, there must exist an F', the inverse of F,

$$D = \underline{F}'(E, \underline{K})$$

which, therefore, has the property that the original data can be recovered from the encrypted data if the value of the key variable originally used is known.

The use of F and F' is valuable only if it is impractical to recover D from E without knowledge of the corresponding K. A great deal of research has been done to develop algorithms which make it virtually impossible to do so, even given the availability of powerful computer tools.

The strength of an encryption algorithm is traditionally evaluated using the following assumptions. First, the algorithm is known to all involved. Second, the analyst has available to him a significant quantity of encrypted data and corresponding cleartext (i.e. the unencrypted text, also called plaintext). He may even have been able to cause messages of his choice to be encrypted. His task is to deduce, given an additional unmatched piece of encrypted text, the corresponding cleartext. All of the matched text can be assumed to be encrypted through the use of the same key which was used to encrypt the unmatched

segment. The difficulty of deducing the key is directly related to the strength of the algorithm.

F is invariably designed to mask statistical properties of the cleartext. Ideally the probability of each symbol of the encrypted character set appearing in an encoded message E is to be equal. Further, the probability distribution of any pair (digram) of such characters is to be flat. Similarly, it is desirable that the n-gram probability distribution be as flat as possible for each n. This characteristic is desired even in the face of skewed distributions in the cleartext, for it is the statistical structure of the input language, as it "shows through" to the encrypted language, which permits cryptanalysis.

The preceding characteristics, desirable from a protection viewpoint, have other implications. In particular, if any single bit of a cleartext message is altered, then the probability of any particular bit being altered in the corresponding message is approximately $\frac{1}{2}$. Conversely, if any single bit in an encrypted message is changed, the probability is approximately $\frac{1}{2}$ that any particular bit in the resulting decrypted message has been changed [FEIS75]. This property follows because of the necessity for flat n-gram distributions. As a result, encryption algorithms are excellent error detection mechanisms, as long as the recipient has any knowledge of

the original cleartext transmission.

The strength of an encryption algorithm is also related to the ratio of the length of the key with the length of the data. Perfect ciphers that completely mask statistical information require keys of lengths equal to the data they encode. Fortunately, currently available algorithms are of such high quality that this ratio can be small; as a result, a key can be often reused for subsequent messages. That is, subsequent messages essentially extend the length of the data. It is still the case that keys need to be changed periodically to prevent the ratio from becoming too small, and, thus, the statistical information available to an analyst too great. The loss of protection which would result from a compromised key is thus also limited.

5.4.2 Public-Key Encryption

Diffie and Hellman [DIFF76b] proposed a variation of conventional encryption methods that may, in some cases, have certain advantages over standard algorithms. In their class of algorithms, there exists

$$E = \underline{F}(D, K),$$

as before, to encode the data, and

$$D = \underline{F}'(E, K')$$

to recover the data. The major difference is that the key

K' used to decrypt the data is not equal to, and is impractical to derive from, the key K used to encode the data. Presumably there exists a pair generator which, on the basis of some input information, produces the matched keys K and K' with high strength (i.e., resistance to the derivation of K' given K , D , and matched $E = F(D, K)$).

Many public-key algorithms have the property that either F or F' can be used for encryption, and both result in strong ciphers. That is, one can encode data using F' , and decode using F . The RSA algorithm is one that has this property [RIVE77a]. The property is useful both in key distribution and "digital signatures" (the electronic analogs of handwritten signatures) and will be assumed here.

The potential value of such encryption algorithms lies in some expected simplifications in initial key distribution, since K can be publicly known; hence the name public-key encryption. There are also simplifications for digital signatures. These issues are examined further in section 5.12 and chapter 7. Rivest et. al. and Merkle and Hellman have proposed actual algorithms which are believed strong, but they have not yet been extensively evaluated [RIVE77a, HELL78].

Much of the remaining material in this chapter is presented in a manner independent of whether conventional- or public-key based encryption is employed. Each case is

considered separately when necessary.

5.5 Error Detection and Duplicate or Missing Blocks

Given the general properties of encryption just described, it is an easy matter to detect (but not correct) errors in encrypted messages. A small part of the message must be redundant, and the receiver must know in advance the expected redundant part of the message. In a block with k check bits, the probability of an undetected error upon receipt of the block is approximately $1/2^k$ for reasonably sized blocks, if the probabilistic assumption mentioned in section 5.4 is valid. For example, if three eight-bit characters are employed as checks, the probability of an undetected error is less than $1/2^{24}$ or $1/10^7$.

In the case of natural language text, no special provisions need necessarily be made, since that text already contains considerable redundancy and casual inspection permits error detection with very high probability. The check field can also be combined with information required in the block for reasons other than encryption. In fact, the packet headers required in most packet switched networks contain considerable highly formatted information, which can serve the check function. For example, duplicate transmitted blocks may occur either because of a deliberate attempt or through abnormal operation of the network switching centers. To detect the duplication, it is

customary to number each block in order of transmission. If this number contains enough bits and the encryption block size matches the unit of transmission, the sequence number can serve as the check field.

Feistel et. al. [FEIS75] describe a variant of this method, called block chaining, in which a small segment of the preceding encrypted block is appended to the current cleartext block before encryption and transmission. The receiver can therefore easily check that blocks have been received in proper order by making the obvious check. However, if the check fails, he cannot tell how many blocks are missing. In both of these cases, once a block is lost and not recoverable by lower level network protocols, some method for reestablishing validity is needed. One method is to obtain new matched keys. An alternative (essential for public-key systems) is to employ an authentication protocol (as described in section 5.11) to choose a new, valid sequence number or data value to restart block chaining.

5.6 Block versus Stream Ciphers

Whether an encryption method is a block or stream cipher affects the strength of the algorithm and has implications for computer use. A stream cipher, in deciding how to encode the next bits of a message, can use the entire preceding portion of the message, as well as the key and the current bits. A block cipher, on the other hand, encodes

each successive block of a message based on that block only and the given key. It is easier to construct strong stream ciphers than strong block ciphers. However, stream ciphers have the characteristic that an error in a given block makes subsequent blocks undecipherable. In many cases, either method may be satisfactory, since lower level network protocols can handle necessary retransmission of garbled or lost blocks. Independent of whether a block or stream cipher is employed, some check data, as mentioned earlier, is still required to detect invalid blocks. In the stream cipher case, when an invalid block is discovered after decoding, the decryption process must be reset to its state preceding the invalid block.

Stream ciphers are less acceptable for computer use in general. If one wishes to be able to update portions of a long encrypted message (or file) selectively, then block ciphers permit decryption, update, and reencryption of the relevant blocks alone, while stream ciphers require reencryption of all subsequent blocks in the stream. So block ciphers are usually preferred. The Lucifer system [FEIS73] is a candidate as a reasonably strong block cipher. Whether or not the National Bureau of Standards' Data Encryption Standard (DES), with its 56 bit keys, is suitably strong is open to debate [DIFF77], but it is being accepted by many commercial users as adequate [NBS77].

5.7 Network Applications of Encryption

Four general uses of encryption having application in computer networks are briefly described in this section. Much of the remainder of the dissertation is devoted to detailed discussions of them.

5.7.1 Authentication

One of the important requirements in computer communications security is to provide a method by which participants in the communication can identify one another in a secure manner. Encryption solves this problem in several ways. First, possession of the right key is taken as *prima facie* evidence that the participant is able to engage in the message exchanges. The transmitter can be assured that only the holder of the key is able to send or receive transmissions in an intelligible way.

Even using secure authentication, one is still subject to the problems caused by lost messages, replayed valid messages, and the reuse of keys for multiple conversations (which exacerbates the replay problem). A general authentication protocol which can detect receipt of previously recorded messages when the keys have not been changed is presented later. The actual procedures by which keys are distributed in the general case is, of course, important, and will be discussed in subsequent sections.

5.7.2 Private Communication

The traditional use of encryption has been in communications where the sender and receiver do not trust the transmission medium, be it be a hand carried note or megabytes shipped over high-capacity satellite channels. This use is crucial in computer networks.

5.7.3 Network Mail

In the private communication function, it is generally understood that first, all parties wishing to communicate are present, and second, they are willing to tolerate some overhead in order to get the private conversation established. A key distribution algorithm involving several messages and interaction with all participants would be acceptable. In the case of electronic mail, which typically involves short messages, it may be unreasonable for the actual transmission to require such significant overhead. Mail should not require that the receiver actually be present at the time the message is sent or received. Since there is no need for immediate delivery, it may be possible to get lower overhead at the cost of increased queueing delays.

5.7.4 Digital Signatures

The goal here is to allow the author of a digitally represented message to "sign" it in such a fashion that the

"signature" has properties similar to an analog signature written in ink for the paper world. Without a suitable digital signature method, the growth of distributed systems may be seriously inhibited, since many transactions, such as those involved in banking, require a legally enforceable contract.

The properties desired of a digital signature method include the following:

1. **Unforgeability:** Only the actual author should be able to create the signature.
2. **Authenticity:** There must be a straightforward way to demonstrate conclusively the validity of a signature in case of dispute, even long after authorship.
3. **No repudiation:** It must not be possible for the author of signed correspondence to subsequently disclaim authorship.
4. **Low cost and high convenience:** The simpler and lower cost the method, the more likely it will be used.

5.8 Minimum Trusted Mechanism; Minimum Central Mechanism

In all of the functions presented in section 5.8, it is desirable that there be a minimum number of trusted mechanisms involved [POPE74b]. This desire occurs because the more mechanism, the greater the opportunity for error,

either by accident or by intention (perhaps by the developers or maintainers). One wishes to minimize the involvement of a central mechanism for analogous reasons. This fear of large complex and central mechanisms is well justified, given the experience of failure of large central operating systems and data management systems to provide a reasonable level of protection against penetration [POPE74a, CARL75]. Kernel-based approaches to software architectures have been developed to address this problem; they have as their goal the minimization of the size and complexity of trusted central mechanisms. For more information about such designs, see [MCCA79, POPE79, DOWN79].

Some people are also distrustful that a centralized governmental communication facility, or even a large common carrier, can ensure privacy and other related characteristics. These general criteria are quite important to the safety and credibility of whatever system is eventually adopted. They also constrain the set of approaches that may be employed.

5.9 Limitations of Encryption

While encryption can contribute in useful ways to the protection of information in computing systems, there are a number of practical limitations to the class of applications for which it is viable. Several of these limitations are discussed below.

5.9.1 Processing in Cleartext

Most of the operations that one wishes to perform on data, from simple arithmetic operations to the complex procedure of constructing indexes to data bases, require that the data be supplied in cleartext. Therefore, the internal controls of the operating system, and to some extent the applications software, must preserve protection controls while the cleartext data is present. While some have proposed that it might be possible to maintain the encrypted data in main memory and have it decrypted only upon loading into CPU registers (and subsequently reencrypted before storage into memory), there are serious questions as to the feasibility of this approach [GAIN77]. The key management facility required is nontrivial, and the difficulties inherent in providing convenient controlled sharing seem forbidding. Another suggestion sometimes made is to use an encoding algorithm which is homomorphic with respect to the desired operations [RIVE78]. Then the operation could be performed on the encrypted values, and the result can be decrypted as before. Unfortunately, known encoding schemes with the necessary properties are not strong algorithms, nor is it generally believed that such methods can be constructed.

Therefore, since data must be processed in cleartext, other means are necessary to protect data from being

compromised by applications software while the data are under control of the operating system, and the remarks in the previous section concerning minimization of these additional means are very important to keep in mind.

5.9.2 Revocation

Keys are similar to simple forms of capabilities, that have been proposed for operating systems [DENN66, FABR74]. They act as tickets and serve as conclusive evidence that the holder may access the corresponding data. Holders may pass keys, just as capabilities may be passed. Methods for selective revocation of access are just as complex as those known for capability systems [FABR74]. The only known method is to decrypt the data and reencrypt with a different key. This action invalidates all the old keys and is obviously not very selective. Hence new keys must be redistributed to all those for whom access is still permitted.

5.9.3 Protection Against Modification

Encryption by itself provides no protection against inadvertent or intentional modification of the data. However, it can provide the means of detecting that modification by including as part of the encrypted data a number of check bits. When decryption is performed, if those bits do not match the expected values, then the data

is known to be invalid.

Detection of modification, however, is often not enough protection. In large data bases, for example, it is not uncommon for very long periods to elapse before any particular data item is referenced. It would only be at that point that a modification would be detected. Error correcting codes could be applied to the data after encryption to provide redundancy. However, these will not be helpful if a malicious user has succeeded in modifying stored data and has destroyed the adjacent data containing the redundancy. Therefore, very high quality recovery software would be necessary to restore the data from (possibly very old) archival records.

5.9.4 Key Storage and Management

Every data item that is to be protected independently of other data items requires encryption by its own key. This key must be stored as long as it is desired to be able to access the data. Thus, to be able to protect a large number of long-lived data items separately, the key storage and management problem becomes formidable. The collection of keys immediately becomes so large that safe system storage is essential. After all, it is not practical to require a user to supply the key when needed, and it is not even practical to embed the keys in applications software, since that would mean the applications software would

require very high quality protection.

The problem of key storage is also present in the handling of removable media. Since an entire volume (tape or disk pack) can be encrypted with the same key (or small set of keys), the size of the problem is reduced. If archival media are encrypted, then the keys must be kept for a long period in a highly reliable way. One solution to this problem would be to store the keys on the units to which they correspond, perhaps even in several different places to avoid local errors on the medium. The keys would have to be protected, of course; a simple way would be to encrypt them with yet a different "master" key. The protection of this master key is absolutely essential to the system's security.

In addition, it is valuable for the access control decision to be dependent on the value of the data being protected, or even on the value of other, related data; salary fields are perhaps the most quoted example. In this case, the software involved, be it applications or system procedures, must maintain its own key table storage in order to examine the cleartext form of the data successfully. That storage, as well as the routines which directly access it, require a high-quality protection mechanism beyond encryption.

Since a separate, reliable protection mechanism seems required for the heart of a multiuser system, it is not clear that the use of encryption (which requires the implementation of a second mechanism) is advisable for protection within the system. The system's protection mechanism can usually be straightforwardly extended to provide all necessary protection facilities.

5.10 System Authentication

Authentication refers to the identification of one member of a communication to the other in a reliable, unforgeable way. In early interactive computer systems, the primary issue was to provide a method by which the operating system could determine the identity of the user who was attempting to log in. Typically, user identification involves supplying confidential parameters, such as passwords or answers to personal questions. There was rarely any concern over the machine identifying itself to the user.

In networks, however, mutual authentication is of interest: each "end" of the channel may wish to assure itself of the identify of the other end. Quick inspection of the class of methods used in centralized systems shows that a straightforward extension is unacceptable. Suppose each participant must send a secret password to the other. Then the first member that sends the password is exposed.

The other member may be an imposter, who has now received the necessary information in order to pose to other nodes as the first member. Extension to a series of exchanges of secret information will not solve the problem; it only forces the imposter into a multistep procedure.

There are a number of straightforward encryption-based authentication protocols which provide reliable mutual authentication without exposing either participant. The methods are robust in the face of all the network security threats mentioned earlier. The general principle involves the encryption of a rapidly changing unique value using a prearranged key and has been independently rediscovered by a number of people [FEIS75, KENT76, POPE78a]. An obvious application for such protocols is to establish a mutually agreed upon sequence number or block chaining initial value that can be used to authenticate communications over a secure channel whose keys have been used before. The sequence number or value should either be one that has not been used before, or should be selected at random, in order to protect against undetected replay of previous messages.

Below is an outline of a simple, general authentication sequence between nodes A and B. At the end of the sequence, A has reliably identified itself to B. A similar sequence is needed for B to identify itself to A. Typically, one expects to interleave the messages of both authentication

sequences.

Assume that in the authentication sequence A uses a secret key associated with itself. The reliability of the authentication depends only on the security of that key. Assume that B holds A's matching key (as well as the matching keys for all other parties to which B might talk).

1. B sends A, in cleartext, a random, unique data item, in this case the current time of day as known to B.
2. A encrypts the received time of day using its authentication key and sends the resulting ciphertext to B.
3. B decrypts A's authentication message, using A's matched key, and compares it with the time of day which B had sent. If they match, then B is satisfied that A was the originator of the message.

This simple protocol exposes neither A nor B if the encryption algorithm is strong, since it should not be possible for a cryptanalyst to be able to deduce the key from the encrypted time of day. This is true even if the cryptanalyst knows the corresponding cleartext time of day. Further, since the authentication messages change rapidly, recording an old message and retransmitting it is not effective.

To use such an authentication protocol to establish a sequence number or initial value for block chaining, A includes that information, before encryption, in its step 2 message to B.

5.11 Key Management

For several participants in a network conversation to communicate securely, it is necessary for them to obtain matching keys to encrypt and decrypt the transmitted data. It should be noted that a matched pair of keys forms a logical channel which is independent of all other such logical channels but as real as any channel created by a network's transmission protocols. Possession of the key admits one to the channel. Without the key the channel is unavailable. Since the common carrier function of the network is to provide many communication channels, how the keys which create the corresponding necessary private channels are supplied is obviously an important matter. The following sections describe various key distribution methods for both conventional and public-key encryption systems.

5.11.1 Conventional Key Distribution

As there are, by assumption, no suitable transmission media for the keys other than the physical network, it will be necessary to devise means to distribute keys over the same physical channels by which actual data is transmitted.

The safety of the logical channels over which the keys are to pass is crucial. Unfortunately, the only available method by which any data, including the keys, can be transmitted in a secure manner is through the very encryption whose initialization is at issue. This seeming circularity is actually easily broken through limited prior distribution of a small number of keys by secure means. The usual approach involves designating a host machine or set of machines [HELL78] on the network to play the role of Key Distribution Center (KDC), at least for the desired connection. It is assumed that a pair of matched keys has been arranged previously between the KDC and each of the potential participants, say A_1, A_2, \dots, A_m . One of the participants, A_i , sends a short message to the KDC asking that matched key pairs be distributed to all the A's, including A_i . If the KDC's protection policy permits the connection, secure messages containing the key and other status information will be sent to each A over the prearranged channels. Data can then be sent over the newly established logical channel. The prearranged key distribution channels carry a low quantity of traffic, and thus, recalling the discussion in section 5.4, the keys can be changed relatively infrequently by other means.

This general approach has many variations to support desirable properties such as a distributed protection policy, integrity in face of crashes, and the like. Some of

these are discussed below.

5.11.2 Centralized Key Control

Perhaps the simplest form of the key distribution method employs a single KDC for the entire network. Therefore n prearranged matched key pairs are required for a network with n distinguishable entities. An obvious disadvantage of this unadorned approach is its affect on network reliability. If communication with the KDC becomes impossible, either because the node on which the KDC is located is down or because the network itself breaks, then the establishment of any further secure communication channels is impossible. If the overall system has been constructed to prevent any interuser communication other than a secure manner, then the entire network eventually stops. This design for distributed systems is, in general, unacceptable except when the underlying communications topology is a star and the KDC is located at the center. Note, however, that this drawback can be fairly easily remedied by the availability of redundant KDCs in case of failure of the main facility.¹¹ The redundant facility can

11. The redundant KDCs form a simple distributed, replicated database, where the replicated information includes private keys and permission controls. However, the database is rarely updated; and when updated, there are no serious requirements for synchronization among the updates. It is not necessary for copies of a key to be updated simultaneously at all sites, for example. Therefore, little additional complexity from the distributed character of the key management function would be expected.

be located at any site which supports a secure operating system and provides appropriate key generation facilities. Centralized key control can quite easily become a performance bottleneck, however.

Needham and Schroeder present an example of how such a KDC would operate [NEED78]. Assume that A and B each have a secret key, K_a and K_b known only to themselves and the KDC. To establish a connection, A sends a request to the KDC requesting a connection to B and includes an identifier (a random number perhaps). The KDC will send back to A: i) a new key K_c to use in the connection, ii) the identifier, iii) a copy of the request, and iv) some information which A can send to B to establish the connection and prove A's identity. That message from the KDC to A is encrypted with A's secret key K_a . Thus, A is the only one who can receive it, and A knows that it is genuine. In addition, A can check the identifier to verify that it is not a replay of some previous request, and can verify that his original cleartext message was not altered before reception by the KDC.

Once A has received this message, A sends to B the data from the KDC intended for B. That data includes the connection key K_c , as well as A's identity, all encrypted by B's secret key. Thus, B now knows the new key, that A is the other party, and that all this came from the KDC.

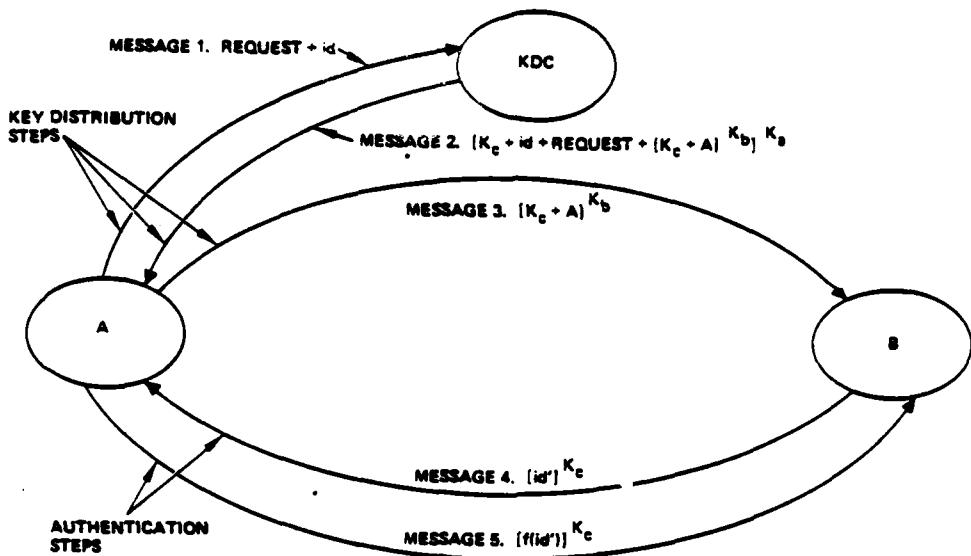


Figure 5-1. Key distribution and conversation establishment: conventional key algorithms. Note: $[ij]$ denotes the cryptogram obtained from the cleartext i , encrypted with key j .

However B does not know that the message he just received is not a replay of some previous message. Thus B must send an identifier to A encrypted by the connection key, upon which A can perform some function and return the result back to B. Now, B knows that A is current, i.e., there has not been a replay of previous messages. Figure 5-1 illustrates the messages involved. Of the five messages, two can be avoided, in general, by storing frequently used keys at the local sites, a technique known as caching.

5.11.3 Fully Distributed Key Control

Here it is possible for every "intelligent" node in the network to serve as a KDC for certain connections. (The

assumption is that some nodes are "dumb", such as terminals or possibly personal computers.) If the intended participants A_1, A_2, \dots, A_m reside at nodes N_1, N_2, \dots, N_m , then only the KDCs at each of those nodes need be involved in the protection decision. One node chooses the key, and sends messages to each of the other KDCs. Each KDC can then decide whether the attempted channel is to be permitted and reply to the originating KDC. At that point the keys would be distributed to the participants. This approach has the obvious advantage that the only nodes which must be properly functioning are those which support the intended participants. Each of the KDCs must be able to communicate with all other KDCs in a secure manner, implying that $n*(n-1)/2$ matched key pairs must have been arranged. Of course, each node needs to store only $n-1$ of them. For such a method to be successful, it is also necessary for each KDC to communicate with the participants at its own node in a secure fashion. This approach permits each host to enforce its own security policy if user software is forced by the local system architecture to use the network only through encrypted channels. This arrangement has appeal in decentralized organizations.

5.11.4 Hierarchical Key Control

This method distributes the key control function among "local," "regional," and "global" controllers. A local

controller is able to communicate securely with entities in its immediate logical locale, that is, for those nodes with which matched key pairs have been arranged. If all the participants in a channel are within the same region, then the connection procedure is the same as for centralized control. If the participants belong to different regions, then it is necessary for the local controller of the originating participant to send a secure message to its regional controller, using a prearranged channel. The regional controller forwards the message to the appropriate local controller, who can communicate with the desired participant. Any of the three levels of KDCs can select the keys. The details of the protocol can vary at this point, depending on the exact manner in which the matched keys are distributed. This design approach obviously generalizes to multiple levels in the case of very large networks. It is analogous to national telephone exchanges, where the exchanges play a role very similar to the KDCs.

One of the desirable properties of this design is the limit it places on the combinatorics of key control. Each local KDC only has to prearrange channels for the potential participants in its area. Regional controllers only have to be able to communicate securely with local controllers. While the combinatorics of key control may not appear difficult enough to warrant this kind of solution, in the subsequent section on levels of integration circumstances

are described in which the problem may be very serious.

The design also has a property not present in either of the preceding key control architectures: local consequences of local failures. If any component of the distributed key control facility should fail or be subverted, then only users local to the failed component are affected. Since the regional and global controllers are of considerable importance to the architecture, it would be advisable to replicate them so that the crash of a single node will not segment the network.

All of these key control methods permit easy extension to the interconnection of different networks, with differing encryption disciplines. The usual way to connect different networks, which often employ different transmission protocols, is to have a single host called a gateway common to both networks [CERF78, BOGG80]. Inter-network data is sent to the gateway, which forwards it toward the final destination. The gateway is responsible for any format conversions, as well as the support of both systems' protocols and naming methods. If the networks' transmissions are encrypted in a manner similar to that described here, then the gateway might be responsible for decrypting the message and reencrypting it for retransmission in the next network. This step is necessary if the encryption algorithms differ, or if there are

significant differences in protocol. If the facilities are compatible, then the gateway can merely serve as a regional key controller for both networks, or even be totally uninvolved.

There are strong similarities among these various methods of key distribution, and differences can be reduced further by designing hybrids to gain some of the advantages of each. Centralized control is a degenerate case of hierarchical control. Fully distributed control can be viewed as a variant of hierarchical control. Each host's KDC acts as a local key controller for that host's entities and communicates with other local key controllers to accomplish a connection. In that case, of course, the communication is direct, without a regional controller required.

5.11.5 Public-Key Based Distribution Algorithms

The class of public key algorithms discussed earlier has been suggested as a candidate for key distribution methods that might be simpler than those described in the preceding sections. Recall that K' , the key used to decipher the encoded message, cannot be derived from K , the key used for encryption, or from matched encrypted and cleartext. Therefore, each user A, after obtaining a matched key pair $\langle K, K' \rangle$, can publicize his key K . Another user B, wishing to send a message to A, can employ the

publicly available key K. To reply, A employs B's public key. At first glance this mechanism seems to provide a simplified way to establish secure communication channels. No secure dialog with a key controller to initiate a channel appears necessary.

The idea is that an automated "telephone book" of public keys could be made available. Whenever user A wishes to communicate with user B, A merely looks up B's public key in the book, encrypts the message with that key, and sends it to B [DIFF76b]. There is no key distribution problem at all. Further, no central authority is required to set up the channel between A and B.

This idea, however, is incorrect: some form of central authority is needed and the protocol involved is no simpler nor any more efficient than one based on conventional algorithms [NEED78]. First, the safety of the public-key scheme depends critically on the correct public key being selected by the sender. If the key listed with a name in the "telephone book" is the wrong one, then the protection supplied by public-key encryption has been lost. Furthermore, maintenance of the (by necessity, machine-supported) book is nontrivial because keys will change, either because of the desire to replace a key which has been used for high amounts of data transmission or because a key has been compromised through a variety of ways. There must

be some source of carefully maintained "books" with the responsibility of carefully authenticating any changes and correctly sending out public keys (or entire copies of the book) upon request.

A modified version of Needham and Schroeder's proposal follows. Assume that A and B each have a public key known to the authority and a private key known only to themselves. Additionally, assume the authority has a public key known to all and a private key known only to the authority.

A begins by sending to the authority a time-stamped message requesting communication with B. The authority sends A the public key of B, a copy of the original request, and the time-stamp, encrypted using the private key of the authority. A can decrypt this message using the public key of the authority and is thus also sure of the source of the message. The time-stamp guarantees that this is not an old message from the authority containing a key other than B's current public key, and the copy of the request permits A to verify that his original cleartext message was not altered.¹²

A can now send messages to B because he knows B's public key. However, to identify himself, as well as to prevent a replay of previous transmissions, A now sends his

12. These initial steps are essentially an adaptation of the authentication protocol given in section 5.11.

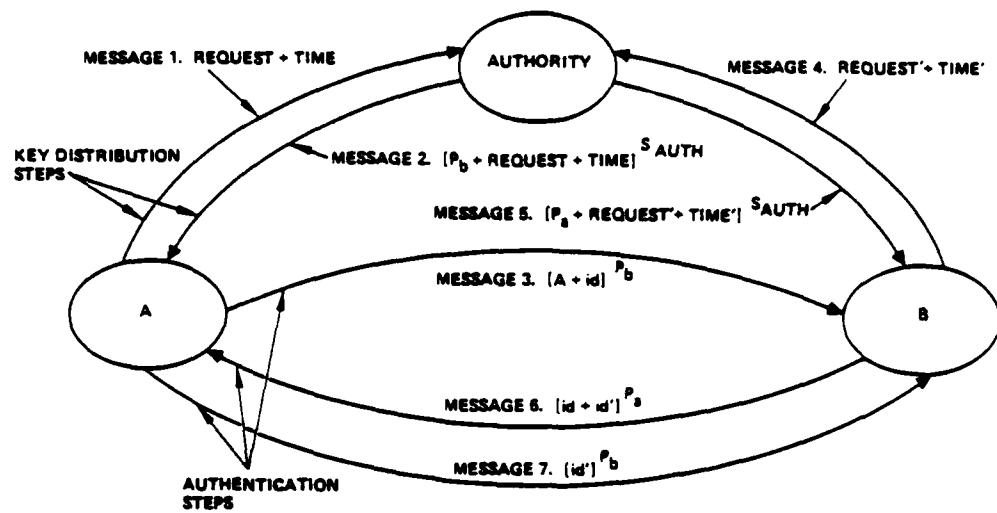


Figure 5-2. Key distribution and conversation establishment: public-key algorithms. Note: P_i is the public key for i ; S_i is the secret key for i .

name and an identifier to B, encrypted in B's public key. B now performs the first two steps above with the authority to retrieve A's public key. Then B sends to A the identifier just received, and an additional identifier, both encrypted with A's public key. A can decrypt that message and is now sure that he is talking to the current B. A must now send back the new identifier to B so that B can be sure he is talking to a current A. These messages are displayed in Figure 5-2. The above protocol contains seven messages, but four of them, those which retrieve the public keys, can be largely dispensed with by local caching of public keys. Thus, as in the conventional key distribution example, three messages are needed.

Some public key advocates have suggested ways other than caching in order to avoid requesting the public key from the central authority for each communication. One such proposal is the use of certificates [KOHN78]. A user can request that his public key be sent to him as a certificate, which is a user/public-key pair, together with some certifying information. For example, the user/public-key pair may be stored as a signed message¹³ from the central authority. When the user wishes to communicate with other users, he sends the certificate to them. They each can check the validity of the certificate, using the certifying information, and then retrieve the public key. Thus the central authority is only needed once, when the initial certificate is requested.

Both certificates and caching have several problems. First, the mechanism used to store the cache of keys must be correct. Second, the user of the certificate must decode it and check it (verify the signature) each time before using it, or must also have a secure and correct way of storing the key. Perhaps most important, as keys change, the cache and old certificates become obsolete. This is essentially the capability revocation problem revisited [REDE74]. Either the keys must be verified (or re-requested) periodically, or a global search must be made whenever invalidating a key. Notice that even with the cache or

¹³. See chapter 7 for a discussion of digital signatures.

certificates, an internal authentication mechanism is still required.

Public-key systems also have the problem that it is more difficult to provide protection policy checks. In particular, conventional encryption mechanisms easily allow protection policy issues to be merged with key distribution. If two users may not communicate, then the key controller can refuse to distribute keys.¹⁴ However, public-key systems imply the knowledge of the public keys. Methods to add protection checks to public key systems add an additional layer of mechanism.

5.11.6 Comparison of Public-Key and Conventional Key Distribution for Private Communication

It should be clear that both of the above protocols establish a secure channel, and that both require the same amount of overhead to establish a connection (three messages). Even if that amount had been different by a message or two, the overhead is still small compared to the number of messages for which a typical connection will be used.

The above protocols can be modified to handle multiple authorities; such modifications have also been performed by

¹⁴. This approach blocks communication if the host operating systems are constructed in such a way as to prohibit cleartext communication over the network.

Needham and Schroeder [NEED78]. Again, the number of messages can be reduced to three by caching.

It should also be noticed that the safety of these methods depends only on the safety of the secret keys in the conventional method, or the private keys in the public-key method. Thus an equivalent amount of secure storage is required.

One might suspect, however, that the software required to implement a public-key authority would be simpler than that for a KDC, and therefore it would be easier to certify its correct operation. If this view were correct, it would make public-key based encryption potentially superior to conventional algorithms, despite the equivalent protocol requirements. It is true that the contents of the authority need not be protected against unauthorized reference, since the public keys are to be available to all, while the keys used in the authentication protocol between the KDC and the user must be protected against reference. However, the standards of software reliability which need to be imposed on the authority for the sake of correctness are not substantially different from those required for the development of a secure KDC. More convincing, all of the KDC keys could be stored in encrypted form, using a KDC master key, and only decrypted when needed. Then the security of the KDC is reduced to protection of the KDC's

master key and of the individual keys when in use. This situation is equivalent to the public-key repository case, since there the private key of the repository must be safely stored and protected during use.

It has also been pointed out that a conventional KDC, since it issued the conversation key, can listen in and in fact generate what appear to be valid messages. Such action cannot be done by the public key repository. This distinction is minor however. Given that both systems require a trusted agent, it is a simple matter to add a few lines of certified correct code to the conventional key agent (the KDC) that destroys conversation keys immediately after distribution. Thus the system characteristics of both conventional and public-key algorithms, as used to support private communication, are more similar than initially expected.

5.12 Levels of Integration

There are many possible choices of endpoints for the encryption channel in a computer network, each with their own trade-offs. In a packet-switched network, one could encrypt each line between two switches separately from all other lines. This is a low-level choice, and is often called link encryption. Instead the endpoints of the encryption channels could be chosen at a higher architectural level: at the host machines which are

connected to the network. Thus the encryption system would support host-host channels, and a message would be encrypted only once as it was sent through the network (or networks) rather than being decrypted and reencrypted a number of times, as implied by the low level choice. In fact, one could even choose a higher architectural level: endpoints could be individual processes within the operating systems of the machines that are attached to the network. If the user were employing an intelligent terminal, then the terminal would be a candidate for an endpoint. This viewpoint envisions a single encryption channel from the user directly to the program with which he is interacting, even though that program might be running on a site other than the one to which the terminal is connected. This high-level choice of endpoints is sometimes called end-to-end encryption.

The choice of architectural level in which the encryption is to be integrated has many ramifications. One of the most important is the combinatorics of key control versus the amount of trusted software.

In general, as one considers higher and higher system levels, the number of identifiable and separately protected entities in the system tends to increase, sometimes dramatically. For example, while there are less than a hundred hosts attached to the Arpanet [ROBE73], at a higher

level there often are over a thousand processes concurrently operating, each one separately protected and controlled. The number of terminals is of course also high. This numerical increase means that the number of previously arranged secure channels—that is, the number of separately distributed matched key pairs—is correspondingly larger. Also, the rate at which keys must be generated and distributed can be dramatically increased.

In return for the additional cost and complexity which results from higher level choices, there can be significant reduction in the amount of software whose correct functioning must be assured. This issue is very important and must be carefully considered. It arises in the following way. When the lowest level (i.e. link encryption) is chosen, the data being communicated exists in cleartext form as it is passed by the switch from one encrypted link to the next. Therefore the software in the switch must be trusted not to intermix packets of different channels. If a higher level is selected, then protection errors in the switches are of little consequence. If the higher level chosen is host to host, however, operating system failures are still serious, because the data exists as cleartext while it is system resident.

In principle then, the highest level integration of encryption is most secure. However, it is still the case

that the data must be maintained in clear form in the machine upon which processing is done. Therefore the more classical methods of protection within individual machines are still necessary, and the value of very high level end-end encryption thereby somewhat lessened. A rather appealing choice of level that integrates effectively with kernel-structured operating system architectures is outlined in the case study in chapter 6.

Another operational drawback to high level encryption should be pointed out. Once the data is encrypted, it is difficult to perform meaningful operations on it. Many front end systems provide such low level functions as packing, character erasures, and transmission on end-of-line or control-character detect. If the data is encrypted when it reaches the front end, then these functions cannot be performed. Any channel processing must be done above the level at which encryption takes place, despite the fact that performance and considerations such as the above sometimes imply a lower level.

5.13 Encryption Protocols

Network communication protocols concern the discipline imposed on messages sent throughout the network to control virtually all aspects of data traffic, both in amount and direction. Choice of protocol has dramatic impacts on the flexibility and bandwidth provided by the network. Since

encryption facilities provide a potentially large set of logical channels, the encryption protocols by which the operation of these channels is managed also has significant impact on system architecture and performance.

There are several important questions which any encryption protocol must answer:

1. How is the initial cleartext/ciphertext/cleartext channel from sender to receiver and back established?
2. How are cleartext addresses passed by the sender around the encryption facilities to the network without providing a path by which cleartext data can be inadvertently or intentionally leaked by the same means?
3. What facilities are provided for error recovery and resynchronization of the protocol?
4. How are channels closed?
5. How do the encryption protocols interact with the rest of the network protocols?
6. How much software is needed to implement the encryption protocols? Does the security of the network depend on this software?

One wishes a protocol which permits channels to be

dynamically opened and closed, allows the traffic flow rate to be controlled (by the receiver presumably), provides reasonable error handling, all with a minimum of mechanism upon which the security of the network depends. The more software involved, the more one must be concerned about the safety of the overall network. Performance resulting from use of the protocol must compare favorably with the attainable performance of the network using other suitable protocols not including encryption. One would prefer a general protocol which could also be added to existing networks, disturbing their existing transmission mechanisms as little as possible. The appropriate level of integration of encryption or the method of key distribution must be considered as well.

Fortunately, the encryption channel can be managed independently of the conventional communication channel, which is responsible for communication initiation and closing, flow control, error handling, and the like. As a result, many protocol questions can be ignored by the encryption facilities and handled by conventional means.

Chapter 6 contains an outline of a complete protocol in order to illustrate the ways in which these considerations interact and the independence that exists. The case considered employs distributed key management and an end-to-end architecture, all added to an existing network.

5.14 Confinement

To confine a program, process, or user means to render it unable to communicate other than through the explicitly controlled paths. Often improper communications are possible through subtle, sometimes timing-dependent channels. As an example, two processes might bypass the controlled channels by affecting each other's data throughput. Although many such improper channels are inherently error prone, the users may employ error detection and correction protocols to overcome that problem.

Unfortunately, the confinement problem in computer networks is particularly difficult to solve because most network designs require some information to be transmitted in cleartext form. This cleartext information, although limited, can be used for the passage of unauthorized information. In particular, the function of routing a message from computer to computer toward its final destination requires that the headers which contain network addresses and control information be in cleartext form, at least inside of the switching centers. A malicious user, cooperating with a penetrator, can send data by the ordering of messages among two communication channels. Even though the data of the communications is encrypted, the headers often are transmitted in cleartext form, unless link encryption is also used to encrypt the entire packet,

including header. In any case, the routing task, often handled in large networks by a set of dedicated interconnected machines which form a subnetwork, requires host addresses in the clear within the switching machines. Thus a penetrator who can capture parts of the subnetwork can receive information. The only solutions to this problem appear to be certification of the secure nature of some parts of the subnetwork and host hardware/software. Work is in progress at the University of Texas on the application of program verification methods to this problem [GOOD77].

Certain confinement problems remain even if certification is applied as suggested. For example, the protocol-implementing software in a given system usually manipulates communications for several users simultaneously. Either this software must be trusted, or data must be encrypted before it reaches this software. Even in this latter case, certain information may be passed between the user and the network software, and thus, potentially, to an unauthorized user. As an example, if a queue is used to hold information waiting to be sent from the user to the network, the user can receive information by noticing the amount drained from this queue by the network software. In almost any reasonable implementation on a system with finite resources, the user will at least be able to sense the time of data removal, if not the amount.

How well current program verification and certification methods apply here is open to question, since these confinement channels are quite likely to exist even in a correct implementation. That is, any feasible design seems to include such channels.

Given the difficulty of confinement enforcement, it is fortunate that most applications do not require it.

Chapter 6 - Network Encryption Protocols

6.1 Network Encryption Protocol Case Study: Private Communication at Process-Process Level

This chapter presents a case study of how encryption was integrated into a real system to recognize the importance of the various issues already presented. The example here was designed and implemented for the Arpanet, and is described in more detail by Popek and Kline [POPE78a]; here is only an outline of the solution in general terms. The goal is to provide secure communication that does not involve application software in the security facilities, nor require trusting that software in order to be assured of the safety of the facilities. It is also desired to minimize the amount of trusted system software.

The protocol provides process-to-process channels and guarantees that it is not possible for application software running within the process to cause cleartext to be transmitted onto the network. Basic operation of the protocol is suggested in Figure 6-1. It is assumed, in keeping with the discussion in chapter 3, that the system software base at each node is a suitably small, secure operating system kernel, which operates correctly.

It is also expected that the amount of software involved in management of the network from the operating

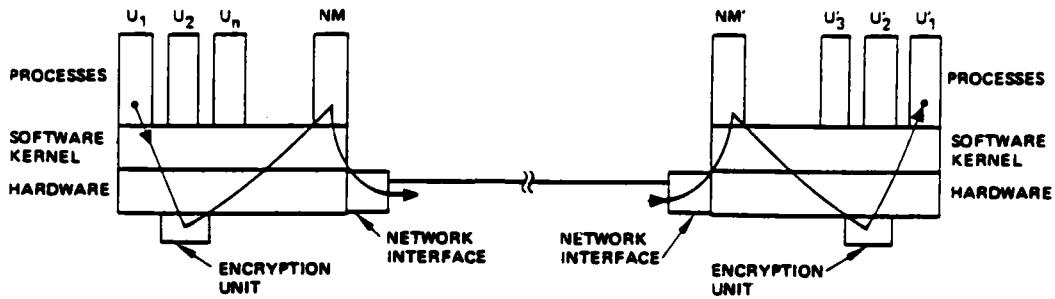


Figure 6-1. Data flow in process-to-process encrypted channels.

system's point of view is substantial; therefore one does not wish to trust its correct operation.¹⁵ Responsibilities of that software include establishing communications channels, supporting retransmission when errors are detected, controlling data flow rates, multiplexing multiple logical channels on the (usually) single physical network connection, and assisting or making routing decisions. The modules which provide these functions are called the network manager (NM).

Assume for the moment that the keys have already been distributed, and logical channels established so far as the network managers are concerned. The operating system nucleus in each case has been augmented with new calls: Encrypt(channel name, data) and Decrypt(channel name, data destination). Whenever a process wishes to send an encrypted block of data, it issues the Encrypt call. The

¹⁵. As an example, in the Arpanet software for the Unix operating system, the network software is comparable in size to the operating system itself.

nucleus takes the data, causes it to be encrypted, and informs the network manager, which can read the block into its workspace. Assuming that the network manager knows what destination site is intended (which it must learn as part of establishing the logical channel), then it can place a cleartext header on the encrypted block and send it out onto the network. The cleartext header is essential so that switching computers which typically make up a network can route the block appropriately.¹⁶

When the block arrives at the destination host computer, the network manager there reads it in and strips off the header. It then tells the kernel the process for which the block is intended. The kernel informs the process, which can issue a Decrypt call, causing the data to be decrypted with the key previously arranged for that process. If this block really is intended for this process (i.e., encrypted with the matching key), then the data is successfully received. Otherwise, decryption with the wrong key yields nonsense. The encrypt and decrypt functions manage sequence numbers in a manner invisible to the user.

Clearly this whole mechanism depends on suitable distribution of keys, together with informing the network

16. Network encryption facilities must, in general, provide some way to supply the header of a message in cleartext, even though the body is encrypted. Otherwise every node on possibly multiple networks has to be able to examine every message; this is not practical.

managers in a coordinated way of the appropriate endpoints of the channel. It is worth noting at this stage that matched keys form a well defined communication channel, and that in the structure just outlined, it is not possible for processes to communicate to the network or the network manager directly; only the encrypt and decrypt functions can be used for this purpose. It is for this latter reason that application software cannot communicate in cleartext over the network, an advantage if that code is not trusted (the usual assumption in military examples).

6.2 Initial Connection

To establish the secure channel, several steps are necessary. The local network manager must be informed with whom the local process wishes to communicate. This would be done by some highly constrained means. The network manager must communicate with the foreign network manager and establish a name for this channel, as well as other state information such as flow control parameters. The network manager software involved need not be trusted. Once these steps are done, encryption keys need to be set up in a safe way.

In the following paragraphs, an outline of how this step would be done employing conventional encryption with fully distributed key management is presented, followed by comments on how it would change if public-key systems were

used.

Assume that there is a kernel-maintained key table which has entries of the form:

foreign host name,
channel name,
sequence number,
local process name,
key.

There are also two additional kernel calls.

Open(foreign process name, local process name, channel name, policy-data) makes the appropriate entry in the key table (if there isn't already one there for the given channel), setting the sequence number to an initial value and sending a message to the foreign kernel of the form <local process name, channel name, policy-data, key>.¹⁷

If there already is an entry in the local key table, it should have been caused by the other host's kernel. In that case Open checks to make sure that the sequence number has been initialized and does not generate a key; rather it sends out the same message, less the key. Close(channel name) deletes the indicated entry in the local key table,

17. The reader will note that the kernel-to-kernel message generated by the Open call must be sent securely and therefore must employ a previously arranged key. The network manager must also be involved, since only it contains the software needed to manage the network.

and sends a message to the foreign kernel to do the same.

The policy data supplied in the Open call, such as classification/clearance information, will be sent to the other site involved in the channel so that it too will have the relevant basis to decide whether or not to allow this channel to be established.

Once both sides have issued corresponding Open calls, the processes can communicate. The following steps illustrate the overall sequence in more detail. The host machines involved are numbered 1 and 2. Process A is at host 1 and B is at host 2. The channel name will be x. The notation NM@i denotes "network manager at site i."

1. A informs NM@1 "connect using x to B@2". This message can be sent locally in the clear. If confinement between the network manager and local processes is important, other methods can be employed to limit the bandwidth between A and the NM.
2. NM@1 sends control messages to NM@2, including whatever host machine protocol messages are required.¹⁸
3. NM@2 receives an interrupt indicating normal message arrival, performs an I/O call to retrieve it, examines the header, determines that it is the recipient and

¹⁸. The host-host protocol messages would normally be sent encrypted using the NM-NM key in most implementations.

processes the message.

4. NM@2 initiates step 2 at site 2, leading to step 3 being executed at site 1 in response. This exchange continues until NM@1 and NM@2 establish a logical channel, using x as their internal name for the channel.
5. NM@1 executes Open(B, A, x, policy-data).
6. In executing the Open, the kernel@1 generates or obtains a key, makes an entry in its key table, and sends a message over its secure channel to kernel@2, which in turn makes a corresponding entry in its table and interrupts NM@2, giving it the triple <B,A,x>.
7. NM@2 issues the corresponding Open(A, B, x, policy-data'). This call interrupts B, and eventually causes the appropriate entry to be made in the kernel table at host 1. The making of that entry interrupts NM@1 and A@1.
8. A and B can now use the channel by issuing successive Encrypt and Decrypt calls.

There are a number of places in the mechanisms just described where failure can occur. If the network software in either of the hosts fails or decides not to open the channel, no kernel calls are involved and standard protocols

operate. (If user notification is permitted, an additional confinement channel is present.) An Open may fail because the name x supplied was already in use, a protection policy check was not successful or because the kernel table was full. The caller is notified. He may try again. In the case of failure of an Open, it may be necessary for the kernel to execute most of the actions of Close to avoid race conditions that can result from other methods of indicating failure to the foreign site.

The encryption mechanism just outlined contains no error correction facilities. If messages are lost, or sequence numbers are out of order or duplicated, the kernel merely notifies the user and network software of the error and renders the channel unusable.¹⁹ This action is taken on all channels, including the host-host protocol channels as well as the kernel-kernel channels. For every case but the last, Closes must be issued and a new channel created via Opens. In the last case, the procedures for bringing up the network must be used.

19. Recall that these sequence numbers are added to the cleartext by the kernel Encrypt call before encryption. They are removed and checked after decryption at the receiving site before delivery to the user. Hence, if desired, sequence numbers can be handled by the encryption unit itself and never seen by kernel software. If such a choice is made, then the conventional network protocols supported by the NM will have to have another set of sequence numbers for error control.

This simple-minded view is acceptable in part because the error rate which the logical encryption channel sees can be quite low. That is, the encryption channel is built on top of lower level facilities supplied by conventional network protocols, some implemented by the NM, which can handle transmission errors (forcing retransmission of errant blocks, for example) before they are visible to the encryption facilities. On highly error prone channels, additional protocol at the encryption level may still be necessary. See [KENT76] for a discussion of resynchronization of the sequencing supported by the encryption channel.

From the protection viewpoint, one can consider the collection of NMs across the network as forming a single (distributed) domain. They may exchange information freely among themselves. No user process can send or receive data directly to or from an NM, except via narrow bandwidth channels through which control information is sent to the NM and status and error information is returned. These channels can be limited by adding parameterized calls to the kernel to pass the minimum amount of data to the NMs and by having the kernel post, to the extent possible, status reports directly to the processes involved. The channel bandwidth cannot be zero, however.

The protocols just presented above in this case study

can also be modified to use public-key algorithms. The kernel, upon receiving the open request, should retrieve the public-key of the recipient. Presumably, the kernel would employ a protocol with the authority to retrieve the public-key and then utilize the authentication mechanisms described in the protocols of chapter 5.

More precisely, in step 6 above, when the kernel receives the Open call, it would retrieve the public-key, either by looking it up in a cache or requesting it from the central authority, or via other methods such as certificates. Once the key is retrieved, the kernel would send a message to the other kernel, over the secure kernel-kernel channel, identifying the user and supplying those policy and authentication parameters required. The other kernel, upon receipt of that message, would retrieve the user's private key (from wherever local user private keys are stored) and continue the authentication sequence.

6.3 System Initialization Procedures

The task of initializing the network software is composed of two important parts. First, it is necessary to establish keys for the secure kernel-kernel channels and the NM-NM channels. Next, the NM can initialize itself and its communications with other NMs. Finally, the kernel can initialize its communications with other kernels. This latter problem is essentially one of mutual authentication

of each kernel with the other member of the pair, and appropriate solutions depend upon the expected threats against which protection is desired.

The initialization of the kernel-kernel channel and NM-NM channel key table entries will require that the kernel maintain initial keys for this purpose. The kernel can not obtain these keys using the above mechanisms at initialization because they require the prior existence of the NM-NM and kernel-kernel channels. Thus, this circularity requires the kernel to maintain at least two key pairs.²⁰ However, such keys could be kept in read only memory of the encryption unit if desired.

The initialization of the NM-NM communications then proceeds as it would if encryption were not present. Once this NM-NM initialization is complete, the kernel-kernel connections could be established by the NM. At this point, the system would be ready for new connection establishment. It should be noted that if desired, the kernels could then set up new keys for the kernel-kernel and NM-NM channels, thus only using the initialization keys for a short time. To avoid overhead at initialization time and to limit the sizes of kernel key tables, NMs probably should only

20. In a centralized key controller version, the only keys which would be needed would be those for the channel between the key controller's NM and the host's NM, and for the channel between the key controller's kernel and the host's kernel. In a distributed key management system, keys would be needed for each key manager.

establish channels with other NMs when a user wants to connect to that particular foreign site, and perhaps should close the NM-NM channel after all user channels are closed.

This case study should serve to illustrate many of the issues present in the design of a suitable network encryption facility.

6.4 Symmetry

The case study portrayed a basically symmetric protocol suitable for use by intelligent nodes, a fairly general case. However, in some instances one of the pair lacks algorithmic capacity, as illustrated by simple hardware terminals or simple microprocessors. Then a strongly asymmetric protocol is required, where the burden of establishing secure communications falls on the more powerful of the pair.

A form of this problem might also occur if encryption is not handled by the system, but rather by the user processes themselves. Then, for certain operations such as sending mail, the receiving user process might not even be present. (Note that such an approach may not guarantee the encryption of all network traffic.) The procedures outlined in the next section are oriented toward reducing the work of one of the members of the communicating pair.

6.5 Network Mail

Recall that network mail may often be short messages, to be delivered as soon as possible to the recipient site and stored there, even if the intended receiver is not currently logged in.

Assume that a user at one site wishes to send a message to a user at another site, but because the second user may not be signed on at the time, a system process (sometimes called a "daemon") is used to receive the mail and deliver it to the user's "mailbox" file for his later inspection. It is desirable that the daemon process not require access to the cleartext form of the mail, for that would require trusting the mail receiver mechanism. This task can be accomplished by sending the mail to the daemon process in encrypted form and having the daemon put that encrypted data directly into the mailbox file. The user can decrypt it when he signs on to read his mail.

In either the conventional or public-key case, the protocols described in chapter 5 can be employed with only slight modifications. In the conventional case, the last two messages, which exchange an identifier to assure that the channel is current, must be dropped (since the recipient may not be present). After the sender requests and gets a conversation key K and a copy of it encrypted with the receiver's secret key, he appends the encrypted mail to the

encrypted conversation key and sends both to the receiver. The receiving mail daemon can deliver the mail and key (both still encrypted), and the intended recipient can decrypt and read it at his leisure.

In the case of public-keys, the sender retrieves the recipient's public-key via an exchange with the repository, encrypts the mail, and sends it to the receiving site. Again the mail daemon delivers the encrypted mail, which can be read later by the recipient since he knows his private-key. Again, the authentication part of the public-key protocol must be dropped. In both of these approaches, since the authentication steps were not performed, the received mail may be a replay of a previous message. If detecting duplicate mail is important, other mechanisms must be used.

Both mechanisms outlined above do guarantee that only the desired recipient of a message will be able to read it. However, as pointed out, the recipient is not guaranteed the identity of the sender. This problem is essentially that of digital signatures, and is discussed in the next chapter.

Chapter 7 - Digital Signatures

7.1 Introduction

The need for digital signatures has by now become apparent. Computer communications are now used for transmission of inter-company memos, contractual agreements, transfers of money, orders for military command and control, and numerous other activities. In many of these applications it is critical that the recipient of a message can rely upon the signature of the author.

7.2 Public-Key Based Methods

At first, it appeared that public-key encryption methods would be superior to conventional ones for use in digital message signatures. The approach, assuming a suitable public-key algorithm, is for the sender to encode the mail with his private key and then send it. The receiver decodes the message with the sender's public key. The usual view is that this procedure does not require a central authority, except to adjudicate an authorship challenge. However, two points should be noted. First, a central authority is needed by the recipient for aid in deciphering the first message received from any given author (to retrieve the corresponding public key, as mentioned in chapter 5). Second, the central authority must keep all old values of public keys in a reliable way to properly

adjudicate conflicts over old signatures (consider the relevant lifetime of a signature on a real estate deed, for example).

Furthermore, and more serious, the unadorned public-key signature protocol just described has an important flaw. The author of signed messages can effectively disavow and repudiate his signatures at any time, merely by causing his secret key to be made public, or "compromised" [SALT78]. When such an event occurs, either by accident or intention, all messages previously "signed" using the given private key are invalidated, since the only proof of validity has been destroyed. Because the private key is now known, anyone could have created any message claimed to have been sent by the given author. None of the signatures can be relied upon.

Hence the validity of a signature on a message is only as safe as the entire future protection of the private key. Further, the ability to remove the protection resides precisely with the individual (the author) who should not hold that right. That is, one important purpose of a signature is to indicate responsibility for the content of the accompanying message in a way that cannot be later disavowed.

The situation with respect to signatures using conventional algorithms might initially appear slightly

better. Rabin [RABI78] proposes a method of digital signatures based on any strong conventional algorithm. Like public-key methods it too requires either a central authority, or an explicit agreement between the two parties involved to get matters going.²¹ Similarly, an adjudicator is required for challenges. Rabin's method, however, uses a large number of keys, with keys not being reused from message to message. As a result, if a few keys are compromised, other signatures based on other keys are still safe. However, this is not a real advantage over public-key methods, since one could readily add a layer of protocol over the public-key method to change keys for each message as Rabin does for conventional methods. One could even use a variant of Rabin's scheme itself with public keys, although it is easy to develop a simpler one.

All of the digital signature methods described or suggested above suffer from the problem of repudiation of signature via key compromise. Rabin's protocol or analogs to it merely limit the damage (or, equivalently, provide selectivity!). It appears that the problem is intrinsic to any approach in which the validity of an author's signature

21. In his paper, Rabin describes an initialization method which involves an explicit contract between each pair of parties that wish to communicate with digitally signed messages. One can easily instead add a central authority to play this role, using suitable authentication protocols, thus obviating any need for two parties to make specific arrangements prior to exchanging signed correspondence.

depends on secret information which can potentially be revealed, either by the author or other interested parties. Surely improvement would be desirable.

7.3 Reliable Digital Signatures

A number of proposals have been made to augment or replace the unadorned approaches just outlined. One, suggested in [KLIN79] employs a network-wide distributed signature facility. Others, based on analogs to notaries public in the paper world, or replicated, trusted archival facilities, provide a dependable time-stamping mechanism so that authors cannot disavow earlier signed correspondence by causing their keys to be revealed.

7.4 Network Registry Based Signatures - A Conventional-Key Approach

The registry solution is based on the obvious approach of interposing some trusted interpretive layer, a secure hardware and/or software "unit", between the author and his signature keys, whatever their form. Then it is a simple matter to organize the collection of units in the network to provide digital signature facilities. Consider all the cooperating units together as a distributed network registry (NR). Some secure communication protocol among the components of the registry is required, but it can be very simple; low-level link-style encryption using conventional

encryption would suffice.

Given that such facilities exist, then a simple implementation of digital signatures which does not require specialized protocols or encryption algorithms is as follows:

1. The author authenticates with a local component of the network registry (NR), creates a message, and hands the message to the NR together with the recipient identifier and an indication that a registered signature is desired.
2. The NR (not necessarily the local component) computes a simple characteristic function of the message, author, recipient, and current time; encrypts the result with a key known only to the NR; and forwards the resulting "signature block" to the recipient. The NR only retains the encryption key employed.
3. The recipient, when the message is received, can ask the NR if the message was indeed signed by the claimed author by presenting the signature block and message. Subsequent challenges are handled in the same way.

Certain precautions are needed to assure the safety of the keys used to encrypt the signature blocks, including the use of different keys between pairs of distributed NR components, and a signature block computation which requires

compromise of multiple components before signature validity is affected. For example, several NR components could each generate fragments of the keys being used. There is not even any need for all NR components to be under control of a single centralized authority, so long as they can all cooperate.

7.5 Notary Public and Archive Based Solutions

Public-key algorithms can provide safe signature methods also. One straightforward method is based on the behavior of notaries public in the paper world. Briefly, there can be a number of independently operating (but perhaps licensed) notary public machines attached to the network. When a signed message has been produced, it can be sent to several of the notary public machines by the author, after the author has signed the message himself. The notary public machine time-stamps the message, signs it itself (thereby encoding it a second time), and returns the result to the author. The author can then put the appropriate cleartext information around the doubly signed correspondence and send it to the intended receiver. He checks the notary's signature by decoding with the notary's public key, then decodes the message using the author's public key. Several notarized copies can be sent, if desired, to increase safety.

The assumption underlying this method is that most of the notaries can be trusted. Since each notary time-stamps its signature, it is not possible for the original author to disavow prior signed correspondence by "losing" his key at a given time. One might think, however, that it is still possible for someone to claim that his key had been revealed sometime in the past without his knowledge and selective messages forged. This problem can be guarded against by having each notary public return a copy of each notarized message to the author's permanent address. (This "patch" of course raises the question of how notaries are kept reliably informed of permanent addresses.)

Each notary is an independent facility, so that no coordination among notaries is required. Of course, if only one notary exists, then the approach is at best no improvement over the scheme presented in the previous section without multiple NR components. Danger of compromise of the notaries' private keys is reduced by the redundant facilities.

A related way to achieve reliable time registration of signed messages is for there to be a number of independent archival sites where either authors or recipients of signed mail may send copies of correspondence to be time-stamped and stored permanently. Of course, the entire message need not be stored; just a characteristic function will do.

Challenges are handled by interrogating the archives. The possibility of an individual's key being compromised and used without his knowledge can be treated in the same way as with notaries public.

7.6 Comparison of Signature Algorithms

The improved conventional-key and public-key based signature algorithms share many common characteristics. They each involve some generally trusted mechanism shared among all those communicating. The safety of signatures still depends on the future protection of keys as before, now including those for the network registry, notaries public, or archive facilities. However, there are several crucial differences from previous proposals. First, the authors of messages do not retain the ability to repudiate signatures at will. Second, the new facilities can be structured so that failure or compromise of several of the components is necessary before signature validity is lost. In the early proposals, a single failure could lead to compromise.

7.7 User Authentication

While digital signatures are important, one must realize that there must still exist a guaranteed authentication mechanism by which an individual is authenticated to the system. Any reasonable communication

system, of course, ultimately requires such a facility, for if one user can masquerade as another, all signature systems will fail. What is required is some reliable way to identify a user sitting at a terminal-some method stronger than the password schemes used today. Perhaps an unforgeable mechanism based on fingerprints or other personal characteristics will emerge.

Chapter 8 - Conclusions

8.1 Computer Security

Early work in computer security mainly involved examining systems for potential weaknesses through the process of penetration analysis [BISB75, CARL75, CARL76, HOLL74, HOLL76]. That work discovered that all systems of that time had numerous security flaws [POPE74c]. Attempts to retrofit security invariably failed. It became clear that constructing secure systems would require new approaches.

Work at UCLA, MITRE, SRI, and other research institutions began developing methodologies for the construction of secure systems. Those methodologies centered around two major areas: security kernels and program verification. The security kernel efforts demonstrated that it was possible to structure systems so that the security relevant modules were not scattered throughout the system, but rather were carefully placed at the heart of the system, not dependent on other code. The verification efforts demonstrated that it was possible to carefully verify that the security kernels enforced the desired security constraints.

This dissertation has presented much of the fruits of that work. Chapter 2 presented a definition for data

security. That definition carefully separates the notions of information flow, proper execution, and security policy. Chapter 3 described the UCLA Data Secure Unix System. The intention here was to give a brief description of the system, explaining the functions necessary in a security kernel, and to provide a brief specification of the implementations of those functions in the UCLA DSU system. Chapter 4 contained a semi-formal intuitive approach to verification of the security properties of a system. The chapter gave examples of the application of the approach to the UCLA DSU system, using the data security definitions of chapter 2. The intent was to provide a technique which allowed people to understand, and thus gain confidence in, security verification.

Chapter 5 explained the problems of network security and, in particular, the problems of using encryption in network security. The use of both conventional and the recently developed public-key encryption systems was explained. Chapter 6 gave an example of how that work might be actually applied and, in fact, the approach given in chapter 6 has been successfully retrofitted into the Arpanet software for the UCLA DSU system. Chapter 7 explained the emerging problem of digital signatures and gave examples of solutions using both conventional and public-key based encryption systems.

8.2 Conclusions and the State of the Art

Several conclusions may be drawn from this research. First, systems possessing a demonstratable level of operating system and network security are feasible today. The UCLA DSU system was the first successful prototype of such systems. The development of production quality systems utilizing that technology is now under way [MCCA79, GOLD79].

Second, the construction of secure systems is still very much an art. While general guidelines have been developed [POPE78c, POPE79, MCCA79], careful engineering is still required. Partly that engineering is in the structuring of the system to remove functions that are not necessary inside the security perimeter. However, much of the art involves the design of the system so that it may be verified and the difficulties in the actual verification effort. It is hoped that this dissertation has aided in the transfer of that technology.

Third, the discussion of network security has been intended to outline the issues in developing secure computer networks, as well as the context in which encryption algorithms will be increasingly used. While public-key algorithms certainly have properties not found in conventional algorithms, it is surprising to note that once the system implications are understood, public-key algorithms and conventional algorithms are largely

equivalent.

Indeed, it is highly unlikely that any given class of encryption algorithms will be sufficient alone to provide the various secure functions which will be desired. Master-key/sub-key relationships, or k-out-of-n systems²² are just two examples. Rather than attempting to develop and evaluate the strength of a new encryption system for each such application, it would be preferable to recognize that a strong extensible system is necessary. Such a system is one for which new characteristics may be easily added, and where the strength of the addition can be demonstrated in a straightforward, incremental manner. Any strong algorithm, either conventional or public-key, can serve as the basis for a strong extensible system when combined with additional trusted management algorithms, expressed either in hardware or software. Examples of such mixed systems were given in chapter 7. In fact, much of the discussion in chapters 5-7 suggest that mixed systems are essential. Once that necessity is recognized, pressure to develop encryption algorithms with special characteristics is lessened; instead, more attention is focussed on the need for strong algorithms in general.

While certain problems in security are now reasonably

²². A k-out-of-n system is one in which any k of a set of n keys are sufficient to decrypt, but it is infeasible to do so with any fewer.

understood, only the surface has been scratched. The technology now exists to construct reasonably secure operating systems. If one assumes that the purpose of a secure network is mainly to provide private pipes, similar to those supplied by common carriers, then general principles by which secure, common carrier based, point to point communication can be provided are reasonably well in hand. As mentioned above for operating systems, in any sophisticated network implementation, there will surely be considerable careful engineering to be done.

One would like, however, to take a rather different view of the nature of the security problem. One would like to be able to construct secure applications and validate the security with only moderate effort. Security controls more sophisticated than simple data security are desired. More general network systems than simple point-to-point channels are required. For example, the goal might be to provide a high level extended machine for the user, in which no explicit awareness of the network is required. The underlying facility is trusted to securely move data from site to site as necessary to support whatever data types and operations are relevant to the user. The facility operates securely and with integrity in the face of unplanned crashes of any nodes in the network; synchronization and security of operations on user meaningful objects (such as Withdrawal from object CheckingAccount) is reliably maintained, using

minimum trusted mechanism. Other higher level security relevant operations beyond digital signatures are provided.

8.3 Suggestions for Future Work

When one takes such high level views of the goals of security, it is clear that much work remains. In the short term, additional research is clearly needed in program verification systems. At the current time, the effort to verify anything about even small programs is quite high. Verification of the security requirements of entire operating systems and general applications is, for all practical purposes, still beyond the state of the art. This must change if verification is to become a practical part of the security certification process.

While the UCLA Data Secure Unix System demonstrated the feasibility of constructing secure operating systems, and work has progressed on the construction of production systems, the methodology and tools available are still inadequate for the task. Further work is clearly needed in this area. As one example, the choice of an implementation language is still not clear: most languages are unsuitable. While the UCLA system was constructed in an extended version of Pascal, that language was a compromise choice, picked mostly to aid in use of an interactive verification system which used Pascal as the input language. Even then, problems were created by the variations between the language

used in the system construction and the language of the verification system.

The UCLA system had a rather novel approach to access control based upon controlling and recording the movement of information [URBA79]. There had been much work in the past exploring access control models, but little work on their relationship with information flow models. What is the relationship? Should future access controls be based on a combined model?

There is considerable work remaining on the issues of network security. With the emergence of high speed local networks, local distributed computer systems are developing. These systems may have tens or hundreds of nodes, all cooperating at a much lower level in the system than examined in this dissertation. How should network security be integrated into these distributed systems? If the systems can be physically secured, for example in a single building, is encryption still relevant?

Finally, as mentioned above, further research is still necessary in encryption. A suitably strong encryption algorithm is needed; there is no suitable methodology to measure the strength of encryption algorithms. Much of the work on algorithmic complexity has dealt with worst case problems-how complex is the algorithm when given worst case inputs? In contrast, encryption algorithms want to be

designed so that they are suitably complex even in the simplest case. In other words, it should be possible to devise an algorithm such that no matter what key is chosen from the (appropriate) key space, the algorithm requires at least some lower bound on cost to break. This problem appears to be quite hard.

Bibliography

ABB074 Abbott, R. P., et. al., A bibliography on computer operating systems security, Lawrence Livermore Laboratory, University of California, The RISOS Project, Report UCRL-51555, April 1974.

ABRA76 Abraham, S. M., A protection design for the UCLA Security Kernel, M.S. Thesis, Computer Science Department, University of California, Los Angeles, 1976.

AH074 Aho, A., Hopcroft, J., and Ullman, J., The design and analysis of computer algorithms, Addison Wesley, Reading, Mass., 1974.

BERS79 Berson, T. A., and Barksdale, G. L., "KSOS - Development methodology for a secure operating system," Proceedings of the National Computer Conference, 48 (1979), AFIPS Press, Arlington Va., 365-371.

BELL73 Bell, D. E., and LaPadula, L. J., Secure computer systems: a refinement of the mathematical model, MTR-2547, Vol. III, MITRE Corp., Bedford, Mass., Dec. 1973.

BELL74 Bell, D. E., and LaPadula, L. J., Secure computer systems: mathematical foundations and model, M74-224, The MITRE Corporation, Bedford, Mass., 1974.

BISB73 Bisbey, R. L. II, and Popek, G. J., "Encapsulation: an approach to operating system security," Proceedings, 1973 ACM Conference, San Diego, Ca., 1973.

BISB75 Bisbey, R. L. II, Popek, G. J., and Carlstedt, J., Protection errors in operating systems: inconsistency of a single data value over time, ISI/SR-75-4, University of Southern California, Information Sciences Institute, Marina Del Rey, Ca., 1975.

BOGG80 Boggs, D., Shoch, J., Taft, E., and Metcalfe, R., Pup: an internetwork architecture, Xerox Palo Alto Research Center, Palo Alto, Ca., 1979.

BRAN73 Branstad, D. K., "Security aspects of computer networks," AIAA Computer Network Systems Conference, AIAA, April, 1973.

BRAN75 Branstad, D. K., "Encryption protection in computer data communications," Proceedings of the Fourth Data Communications Symposium, 1975, 8-1-8-7.

BROW76 Browne, P. S., "Computer security - A survey," Proceedings of the National Computer Conference, 45 (1976), AFIPS Press, Arlington, Va., 53-63.

CARL75 Carlstedt, J., Bisbey, R., and Popek, G., Pattern directed protection evaluation, Report ISI/RR-75-31, University of Southern California, Information Sciences Institute, Marina Del Rey, California, 1975.

CARL76 Carlstedt, J., Protection errors in operating systems: validation of critical conditions, ISI/SR-76-5, University of Southern California, Information Sciences Institute, Marina Del Rey, Ca., 1976.

CERF78 Cerf, V., and Kirstein, P., "Issues in packet-network interconnection," Proceedings of the IEEE, 66, 11 (Nov. 1978), 1386-1408.

COHE78 Cohen, E., "Information transmission in sequential programs," Foundations of Secure Computing, R. DeMillo, et. al., Eds, Academic Press, New York, 1978.

CONN74 Conn, R. W., and Yamamoto, R. H., "A model highlighting the security of operating systems," Proceedings, 1974 ACM Conference, San Diego, Ca., 1974.

COSC78 "Conversations on secure computation," Foundations of Secure Computing, R. DeMillo, et. al., Eds, Academic Press, New York, 1978.

DEC75 PDP 11/45 Processor Handbook, Digital Equipment Corporation, Maynard, Mass., 1975.

DENN66 Dennis, J., and Van Horn, E., "Programming semantics for multiprogrammed computations," Communications of the ACM, 9, 3 (Mar. 1966), 143-155.

DENN75 Denning, D. E., Secure information flow in computer systems, Ph.D. thesis, CSD TR 145, Purdue University, May 1975.

DIFF76a Diffie, W., and Hellman, M., "Multiuser cryptographic techniques," Proceedings of the National Computer Conference, 45 (1976), AFIPS Press, Arlington, Va., 109-112.

DIFF76b Diffie, W., and Hellman, M., "New directions in cryptography," IEEE Transactions on Information Theory, IT-22, 11 (November, 1976), 644-654.

DIFF77 Diffie, W., and Hellman, M., "Exhaustive cryptanalysis of the NBS data encryption standard," Computer, 10, 6 (Jun. 1977), 74-84.

DIFF79 Diffie, W., and Hellman, M., "Privacy and authentication: an introduction to cryptography," Proceedings of the IEEE, 67, 3 (Mar. 1979), 397-427.

DOWN77 Downs, D., and Popek, G. J., "A kernel design for a secure data base management system," Proceedings of the Third International Conference on Very Large Data Bases, Tokyo, Japan, October, 1977, 507-514.

DOWN79 Downs, D., and Popek, G., "Data base system security and Ingres," Proceedings of the conference on Very Large Data Bases, 1979, Rio De Janiero.

EVAN74 Evans, A., Kantrowitz, W., and Weiss, E., "A user authentication system not requiring secrecy in the computer," Communications of the ACM, 17, 8 (Aug. 1974), 437-442.

FABR74 Fabry, R., "Capability based addressing," Communications of the ACM, 17, 7 (Jul. 1974), 403-411.

FEIE79 Feiertag, R., and Neumann, P., "The foundations of a provably secure operating system (PSOS)," Proceedings of the National Computer Conference, 48 (1979), AFIPS Press, Arlington, Va., 329-334.

FEIS73 Feistel, H., "Cryptography and computer privacy", Scientific American, 228, 5 (May 1973), 15-23.

FEIS75 Feistel, H., Notz, W. A., and Smith, J. L., "Some cryptographic techniques for machine to machine data communications," Proceedings of the IEEE, 63, 11 (Nov. 1975), 1545-1554.

GAIN77 Gaines, R. S., private communication, September, 1977.

GOLD79 Gold, B. D., Linde, R. R., Peeler, R. J., Schaefer, M., Scheid, J. F., Ward, P. D., "A security retrofit of VM/370," Proceedings of the National Computer Conference, 48 (1979), AFIPS Press, Arlington Va., 335-344.

GOOD77 Good, D. I., "Constructing verified and reliable communications processing systems," ACM Software Engineering Notes, 2-5, Oct. 77.

GRAH72 Graham, G. S., and Denning, P. J., "Protection - principles and practice," Spring Joint Computer Conference, AFIPS Press, Arlington, Va., May 1972.

HELL78 Hellman, M. E., "Security in communications networks," Proceedings of the National Computer Conference, 47 (1978), AFIPS Press, Arlington, Va., 1131-1134.

HOFF72 Hoffman, L. J., "Computers and privacy: a survey," Computing Surveys, 1, 2 (June 1969).

HOLL74 Hollingsworth, D., Glaseman, S., and Hopwood, M., Security test and evaluation tools: an approach to operating system security analysis, The Rand Corporation, Sept. 1974.

HOLL76 Hollingsworth, D., and Bisbey, R. L. II, Protection errors in operating systems: allocation/deallocation residuals, June 1976.

KAMP77 Kampe, M., Kline, C. S., Popek, G. J., and Walton, E. J., The UCLA Data Secure Operating System Prototype, Computer Science Department, University of California, Los Angeles, Technical Report 77-3, July 1977.

KEMM78 Kemmerer, R. A., A proposal for the formal verification of the security properties of the UCLA Secure UNIX Operating System Kernel, Computer Science Department, University of California, Los Angeles, Technical Report 78-1 (UCLA-ENG-7810), February 1978.

KEMM79 Kemmerer, R. A., Formal verification of the UCLA Security Kernel: Abstract model, mapping functions, theorem generation, and proofs, Ph.D. thesis, UCLA-ENG-7956, University of California at Los Angeles, 1979.

KENT76 Kent, S., Encryption-based protection protocols for interactive user-computer communication, Technical Report 162, M.I.T. Laboratory for Computer Science, May, 1976.

KIMB75 Kimbleton, S. R., and Schneider, G. M., "Computer communications networks: approaches, objectives and performance considerations," Computing Surveys, 7, 3 (Sept. 1975), 129-173.

KLIN77 Kline, C. S., and Popek, G. J., Encryption in computer network security, Computer Science Department, University of California, Los Angeles, Technical Report 77-2, April 1977.

KLIN79 Kline, C. S., and Popek, G. J., "Public key vs. conventional key encryption", Proceedings of the National Computer Conference, 48 (1979), AFIPS Press, Arlington, Va., 831-837.

KOHN78 Kohnfelder, L., Towards a practical public-key cryptosystem, Bachelor of Science Thesis, Massachusetts Institute of Technology, 1978.

LAMP73 Lampson, B. W., "A note on the confinement problem," Communications of the ACM, 16, 10 (Oct. 1973), 613-615.

LENN78 Lennon, R. E., "Cryptography architecture for information security," IBM Systems Journal, IBM, 17, 2 (1978), 138-150.

LIND76 Linden, T. A., "Operating system structures to support security and reliable software," Computing Surveys, 8, 4 (Dec. 1976), 409-445.

MATY78 Matyas, S. M., and Meyer, C. H., "Generation, distribution, and installation of cryptographic keys," IBM Systems Journal, IBM, 17, 2 (1978), 126-137.

MCCA79 McCauley, E. J., and Drongowski, P. J., "KSOS - the design of a secure operating system," Proceedings of the National Computer Conference, 48 (1979), AFIPS Press, Arlington Va., 345-353.

MENA79 Menasce, D. A., Rudisin, G. J., Popek, G. J., and Kline, C. S., A proposed architecture for the distributed secure system base, Computer Science Department, University of California, Los Angeles, Technical Report 79-10 (UCLA-ENG-7957), September 1979.

MERK78 Merkle, R., "Secure communication over insecure channels," Communications of the ACM, 21, 4 (Apr. 1978), 294-299.

MERK79 Merkle, R., Secrecy, authentication, and public key systems, Ph.D. thesis, Information Systems Laboratory, Technical Report 1979-1, Stanford University, Stanford, Ca., 1979.

MEYE73 Meyer, C. H., "Design considerations for cryptography," Proceedings of the National Computer Conference, 42 (1973), AFIPS Press, Arlington, Va., 603-606.

MICH79 Michelman, E. H., "The design and operations of public-key cryptosystems," Proceedings of the National Computer Conference, 48 (1979), AFIPS Press, Arlington Va., 305-311.

NBS77 National Bureau of Standards, Data encryption standard, Federal Information Processing Standards Publication 46, 1977.

NBS78a National Bureau of Standards, Design alternatives for computer network security, National Bureau of Standards, Special Publication 500-21, 1, 1978.

NBS78b National Bureau of Standards, The network security center: A system level approach to computer security, National Bureau of Standards, Special Publication 500-21, 1, 1978.

NEED78 Needham, R., and Schroeder, M., "Using encryption for authentication in large networks of computers", Communications of the ACM, 21, 12 (Dec. 1978), 993-999.

NEUM77 Neumann, P. G., Boyer, R. S., Feiertag, R. J., Levitt, K. N., and Robinson, L., A provably secure operating system: the system, its applications, and proofs, SRI International, Menlo Park, Ca., 1977.

NEUM78 Neumann, P. G., "Computer system security evaluation," Proceedings of the National Computer Conference, 47 (1978), AFIPS Press, Arlington Va., 1087-1095.

PADL79 Padlipsky, M. A., Biba, K. J., and Neely, R. B., "KSOS - computer network applications," Proceedings of the National Computer Conference, 48 (1979), AFIPS Press, Arlington Va., 373-381.

POPE73 Popek, G. J., "Correctness in access control," Proceedings of the ACM National Conference, Atlanta, Georgia, 1973, 236-241.

POPE74a Popek, G. J., "Protection structures," IEEE Computer, (Jul. 1974), 22-33.

POPE74b Popek, G. J., "A principle of kernel design," Proceedings of the National Computer Conference, 43 (1974), AFIPS Press, Arlington, Va., 977-978.

POPE74c Popek, G. J., and Kline, C. S., "Verifiable secure operating system software," Proceedings of the National Computer Conference, 43 (1974), AFIPS Press, Arlington, Va., 145-151.

POPE74d Popek, G. J., and Kline, C. S., "The design of a verified protection system," Proceedings of the IRIA International Workshop on Protection in Operating Systems, Rocquencourt, France, August, 1974, 183-196.

POPE75a Popek, G. J., and Kline, C. S., "A verifiable protection system," Proceedings of the 1975 International Conference on Reliable Software, Los Angeles, Ca., April 21-23, 1975, 294-304.

POPE75b Popek, G. J., and Kline, C. S., "The PDP-11 virtual machine architecture: a case study," Proceedings of the Fifth Symposium on Operating Systems Principles, Austin, Texas, October 1975.

POPE76 Popek, G. J., and Farber, D. A., "On computer security verification," Twelfth IEEE Computer Society International Conference: Compcon 76, San Francisco, Ca., Feb. 1976, 140-145.

POPE77a Popek, G. J., Horning, J. J., Lampson, B. W., Mitchell, J. G., and London, R. L., "Notes on the design of EUCLID," Proceedings of an ACM Conference on Language Design for Reliable Software, Raleigh, North Carolina, March, 1977, 11-18. Also in SIGPLAN Notices, 12, 3 (Sept. 1977).

POPE77b Popek, G. J., and Kline, C. S., "Encryption protocols, public key algorithms and digital signatures in computer networks," Foundations of Secure Computing, R. DeMillo, et. al., eds., Academic Press, New York, 1978, 133-153.

POPE78a Popek, G. J., and Kline, C. S., "Design issues for secure computer networks", Operating Systems, An Advanced Course, R. Bayer, R. M. Graham, G. Seegmuller, Eds., Springer-Verlag, New York, 1978

POPE78b Popek, G. J., and Farber, D. A., "A model for verification of data security in operating systems," Communications of the ACM, 21, 9 (Sept. 1978), 737-749.

POPE78c Popek, G. J., and Kline, C. S., "Issues in kernel design," Proceedings of the National Computer Conference, 47 (1978), AFIPS Press, Arlington, Va., 1079-1086.

POPE78d Popek, G. J., Kampe, M., Kline, C. S., Stoughton, A. H., Urban, M. P., and Walton, E. J., UCLA Data Secure Unix - a secure operating system: software architecture, Technical Report 78-7 (UCLA-ENG-7854), Computer Science Department, University of California, Los Angeles, 1978.

POPE79 Popek, G. J., Kampe, M., Kline, C. S., Stoughton, A., Urban, M., and Walton, E. J., "UCLA Secure Unix," Proceedings of the National Computer Conference, 48 (1979), AFIPS Press, Arlington, Va., 355-364.

RABI78 Rabin, M., "Digitalized signatures", Foundations of Secure Computing, R. DeMillo, et. al., Eds, Academic Press, New York, 1978.

RAND80 Random House, The Random House College Dictionary, Jess Stein, Ed., Random House, New York, N. Y., 1980.

REDE74 Redell, D. D., and Fabry, R. S., "Selective revocation of capabilities", Proceeding of the IRIA Conference on Protection in Operating Systems, Rocquencourt, France, August, 1974.

RIVE77a Rivest, R. L., Shamir, A., and Adleman, L., A method for obtaining digital signatures and public-key cryptosystems, Technical Memo LCS/TM82, M.I.T. Laboratory for Computer Science, Cambridge, Mass., April 4, 1977 (Revised August 31, 1977).

RIVE77b Rivest, R., private communications, 1977.

RIVE78 Rivest, R. L., Shamir, A., and Adleman, L., "On digital signatures and public key cryptosystems," Communications of the ACM, 21, 2 (Feb 1978), 120-

ROBE73 Roberts, L., and Wessler, B., "The ARPA Network," Computer Communication Networks, Prentice-Hall, 1973, 485-499.

ROBI75 Robinson, L., Neumann, P. G., Levitt, K. N., and Saxena, A. R., "On attaining reliable software for a secure operating system," 1975 International Conference on Reliable Software, Los Angeles, Ca., April 1975, 267-284.

SALT74 Saltzer, J. H., "Ongoing research and development on information protection," ACM Operating Systems Review, July 1974.

SALT75 Saltzer, J. H., and Schroeder, M. D., "The protection of information in computer systems," Proceedings of the IEEE, 63, 9 (Sept. 1975), 1278-1308.

SALT78 Saltzer, J., "On digital signatures", ACM Operating Systems Review, 12, 2 (Apr. 1978), 12-14.

SEND78 Sendrow, M., "Key management in EFT environments," Proceedings of COMPCON, 1978, 351-354.

SIMM79 Simmons, G. J., "The asymmetric encryption/decryption channel," Computing Surveys, 11, 4 (Dec. 1979), 308-330.

URBA79 Urban, M. P., The design and implementation of a file policy manager for the UCLA Data Secure Unix System, M.S. Thesis, Computer Science Department, University of California, Los Angeles, 1979.

WALK77 Walker, B. J., Verification of the UCLA Security Kernel: Data Defined Specifications, M. S. Thesis, Computer Science Department, University of California, Los Angeles, 1977.

WALK80 Walker, B. J., Kemmerer, R. A., and Popek, G. J., "Specification and verification of the UCLA Unix Security Kernel," Communications of the ACM, 23, 2 (Feb. 1980), 118-131.

WALT75 Walton, E. J., The UCLA Security Kernel, M.S. Thesis, Computer Science Department, University of California, Los Angeles, June 1973.

WEST70 Westin, A. F., Privacy and freedom, Atheneum Press, New York, 1970.

WOOD77 Woodward, J. P. L., and Nibaldi, G. H., A kernel-based secure UNIX design, MTR-3499, The MITRE Corporation, Bedford, Mass., 1977.

WOOD79 Woodward, J. P. L., "Applications for multilevel secure operating systems," Proceedings of the National Computer Conference, 48 (1979), AFIPS Press, Arlington Va., 319-328.

WULF74 Wulf, W. A., et. al., "HYDRA: the kernel of a multiprocessor operating system," Communications of the ACM, 17, 6 (June 1974), 337-345.

WULF76 Wulf, W. E., London, R. L., and Shaw, M., Abstraction and verification in Alphard: introduction to language and methodology, ISI/RR-76-46, University of Southern California, Information Sciences Institute, Marina del Rey, Ca., June 1976.